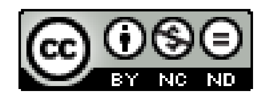
Eugenia Bahit

Curso: Python para Principiantes

www.eugeniabahit.com





Comparte el conocimiento

Eres libre de:

- **Copiar**, **distribuir** y **compartir** este libro

Bajo las siguientes condiciones:

- Reconocer y respetar la autoría de la obra
- No hacer uso comercial de ella
- No alterar el contenido



http://www.safecreative.org/work/1207302042960



Descarga todos los ejemplos de este libro y los talleres del curso, ingresando en

http://curso-python.eugeniabahit.com/sources.tar.gz

00 Tabla de Contenidos (índice)

01 Preparación del entorno de desarrollo

Comenzaremos instalando todo nuestro sistema, para crear un entorno de desarrollo propicio, para trabajar con Python. A tal fin, nos valdremos de las siguientes herramientas y tecnologías:

1. Sistema Operativo GNU/Linux: Ubuntu 11.10 (o superior)
2. Python 2.7
3. iPython (Shell interactivo mejorado)
4. Ninja-IDE (IDE de desarrollo)
5. Bazaar (Sistema de Control de Versiones distribuido)

Introducción al Sistema Operativo GNU/Linux

Antes de comenzar, intentaremos establecer una diferencia, entre los término "Linux" y "GNU/Linux", a fin de saber de qué estamos hablando con exactitud, en cada caso.

Linux, es un kernel, es decir, el núcleo de un Sistema Operativo, mientras que **GNU/Linux, el Sistema Operativo que utiliza el Kernel Linux** como núcleo, creado, difundido y promovido a través del **Proyecto GNU**, por la **Free Software Foundation**, organización sin fines de lucro, fundada por Richard Stallman, principal precursor del Software Libre.

El Kernel Linux, parte fundamental del Sistema Operativo, fue desarrollado por **Linus Torvals**, utilizando como modelo a UNIX. Una de las diferencias fundamentales entre los núcleos Linux y UNIX, es que el primero, es Software Libre, mientras que el segundo no lo es.

Por otra parte, mientras existe un único Kernel Linux (con versiones diferentes), existen decenas y hasta cientos de **distribuciones GNU/Linux**, es decir, diferentes Sistemas Operativos basados en el Kernel Linux, entre las cuales se destacan: **Debian**, **Ubuntu**, **Kubuntu**, **Fedora**, **Gentoo**, **Slackware**, **CentOS**, **ArchLinux**, **Asturix**, entre otros cientos.

Más información al respecto, puede encontrarse en:

- Sitio Web de la **Free Software Foundation**: www.fsf.org
- Sitio Web del **Proyecto GNU**: www.gnu.org
- Sitio Web del **Kernel Linux**: http://www.kernel.org/
- Sitio Web de la **Linux Foundation**: http://www.linuxfoundation.org/
- Introduccion al software libre (Universitat Obierta de Catalunya)
- Sistema operativo gnu linux basico (Universitat Obierta de Catalunya)

Instalación de Ubuntu GNU/Linux en Windows

Si eres usuario de Windows y deseas conservar tu Sistema Operativo actual, **puedes descargar Ubuntu Windows Installer** desde el sitio Web oficial de Canonical (empresa que desarrolla y mantiene Ubuntu) en la siguiente URL: http://www.ubuntu.com/download/ubuntu/windows-installer

Ubuntu Windows Installer se instalará desde el propio MS Windows© como si fuese un Software más, permitiéndote iniciar tu ordenador con Ubuntu o MS Windows© según

elijas.

Para instalar Ubuntu Windows Installer, **sigue las instrucciones de los pasos 2 y 3 de la URL de descarga**, las cuales podrás visualizar pulsando el botón “Show me how” de cada uno de los pasos.

Instalación de Ubuntu GNU/Linux como único Sistema Operativo

Para instalar Ubuntu como único Sistema Operativo, sigue los siguientes pasos:

1. ingresa en http://www.ubuntu.com/download/ubuntu/download
2. En el paso 1, selecciona la versión de Ubuntu que deseas descargar. Para procesadores de un solo núcleo, selecciona la versión 10.04 LTS. Para procesadores más modernos, puedes seleccionar la última versión (versión que aparece seleccionada por defecto en el desplegable de versiones). Si tienes dudas sobre si elegir la versión para 32 o 64 bits, elige la de 32-bits. Pulsa el botón “Start download” y aguarda a que se descargue el archivo.
3. Una vez descargado el archivo, podrás quemarlo en un CD/DVD o un Pendrive USB. En el paso 2 de la URL de descarga, selecciona CD o USB stick según tus preferencias y el Sistema Operativo desde el cual harás la copia (Windows o Mac). Pulsa el botón “show me how” y sigue las instrucciones de quemado.
4. A continuación, salta al paso 4 del sitio de descarga (el 3 es solo para probar Ubuntu sin instalarlo); pulsa el botón “show me how” y sigue las instrucciones para instalar Ubuntu en tu ordenador.

Instalando Python

Una vez que hayas instalado tu distribución GNU/Linux, ya tendrás Python instalado en tu sistema.

Para comprobarlo, abres una terminal (presiona Alt + F4 y luego escribe en el campo de búsqueda gnome-terminal) y escribe python como se muestra a continuación:

```
eugenia@cochito:~$ python
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Lo que verás en pantalla, es el Shell interactivo de Python. **Para salir del Shell interactivo, pulsa las teclas Ctrl + D.**

```
eugenia@cochito:~$ python
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hola Mundo!"
Hola Mundo!
>>>
```

Si en lugar del Shell interactivo, ves un mensaje de error similar a “python: orden no encontrada”, deberás seguir los siguientes pasos para instalarlo:

Actualiza la lista de los repositorios:

```
eugenia@cochito:~$ sudo apt-get update
```

Actualiza el Sistema Operativo:

```
eugenia@cochito:~$ sudo apt-get upgrade
```

Instala Python:

```
eugenia@cochito:~$ sudo apt-get install python2.7
```

SOBRE LOS COMANDOS
sudo: te convierte en super usuario. Único usuario que tiene permisos para instalar paquetes en tu sistema operativo.
apt-get: es la utilidad para manejar paquetes en distribuciones GNU/Linux basadas en Debian. Alternativamente, puedes utilizar

el comando **aptitude** en vez de apt-get.
update: opción de apt-get que sincroniza los archivos del índice de paquetes con los repositorios oficiales (dicho de otra forma, obtiene un índice de actualizaciones)
upgrade: opción de apt-get que actualiza el sistema.
install: es la opción de apt-get que indica que se instalarán uno o más paquetes

Instalación de un Shell interactivo mejorado

Python trae por defecto su propio Shell interactivo, el cuál nos permite escribir código Python y ejecutarlo. Sin embargo, tenemos la opción de contar con un Shell interactivo mejorado, que entre otras ventajas sobre el shell nativo, podemos encontrar números de línea, sangrado automático, etc.

iPython, es el Shell interactivo que elegiremos. Para instalarlo, ejecuta la siguiente orden desde una terminal:

```
eugenia@cochito:~$ sudo apt-get install ipython
```

Para ejecutar el nuevo shell interactivo, solo deberás escribir el comando ipython:

```
eugenia@cochito:~$ ipython
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
Type "copyright", "credits" or "license" for more information.

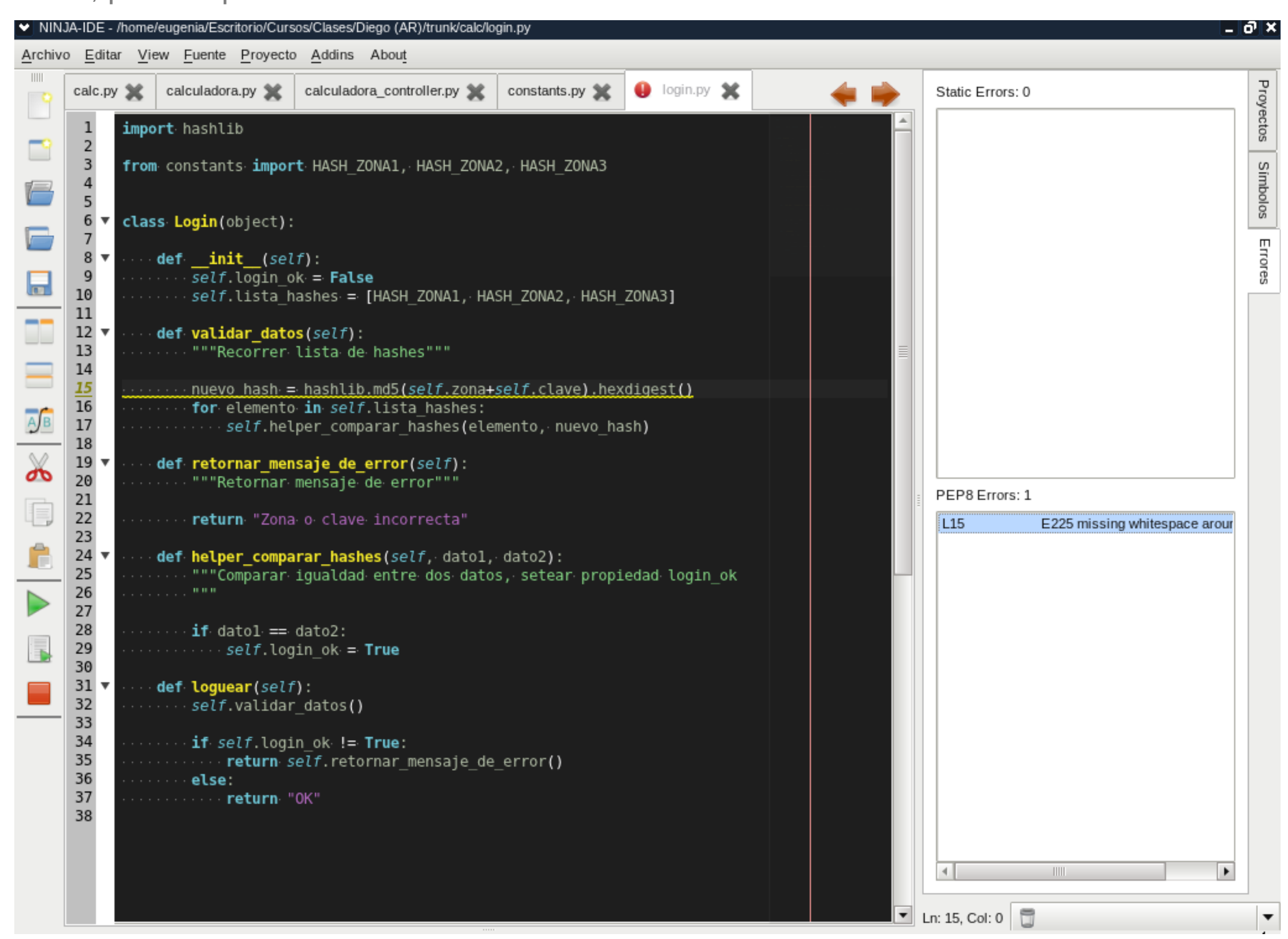
IPython 0.10.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: def mi_funcion():
   ...:     num = raw_input("Numero: ")
   ...:     if num < 10:
   ...:         print "El numero ingresado es menor que 10"
   ...:
   ...:     else:
   ...:         print "El numero ingresado NO es menor que 10"
   ...:
   ...:

In [2]: mi_funcion()
Numero: 154
El numero ingresado NO es menor que 10
```

Ninja-IDE

Ninja-IDE es un Entorno Integrado de Desarrollo[1] que nos permitirá, crear proyectos en Python, al tiempo de ir ejecutando nuestros códigos y corrigiendo eventuales errores que éstos, puedan presentar.



Para instalar Ninja-IDE en tu ordenador, desde la terminal, ejecuta los siguientes comandos:

1) Agrega el PPA de Ninja-IDE:

```
sudo apt-add-repository ppa:ninja-ide-developers/daily
```

2) Sincroniza el índice de actualizaciones:

```
sudo apt-get update
```

3) Instala Ninja-IDE:

```
sudo apt-get install ninja-ide
```

1 http://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

Instalación de Bazaar

Bazaar es un sistema de control de versiones distribuido, que nos permitirá ir manteniendo el control cambios sobre nuestros archivos, centralizándolos en un repositorio.

Un Repositorio es un espacio destinado a almacenar información digital. En nuestro caso, lo que se almacenará en ese repositorio, serán los archivos -código fuente, *tarballs*, binarios, etc- de las aplicaciones y ejercicios que iremos *codeando* a lo largo del curso.

Las ventajas principales de utilizar un SCV, son:

- Espacio de **almacenamiento centralizado** de, principalmente, el código fuente de la aplicación así como scripts de construcción -en el caso de aplicaciones que requieran ser compiladas o simplemente, necesiten realizar configuraciones especiales, ya sea tanto para continuar desarrollándolas como para ejecutarlas-.
- Para ser efectivos, deben llevar un **control histórico de cambios** que se vayan efectuando en los archivos -preferentemente automático-, permitir el establecimiento de tags -etiquetas- que ayuden a identificar diferentes releases -versiones-.

Los Sistemas de Control de Versiones (SCV) pueden agruparse en dos tipos:

- **Centralizados:**

  un único repositorio centralizado administrado por un solo responsable.
- **Distribuidos (recomendados):**

  donde existe un repositorio central que cada usuario podrá clonar para obtener su propio repositorio -local- e interactuar con con otros repositorios locales.

Entre los **SCV distribuidos** podemos destacar excelentes alternativas **GPL** (Software Libre), como es el caso de -entre otros-, **Git** (de Linus Torvalds, creador del Kernel Linux en el que se basa el Sistema Operativo GNU/Linux), **Mercurial** (desarrollado en Python y C) o el magnífico **Bazaar**, nacido a partir de GNUArch y desarrollado íntegramente en Python por Martin Pool, con el patrocinio de Canonical y **elegido en este curso**.

> Una gran **ventaja de los SCV** es que **permiten a varios programadores trabajar simultáneamente sobre los mismos archivos**, impidiendo que el trabajo de uno, pise al trabajo de otro.

Los SCV pueden utilizarse tanto a través de línea de comandos, como de aplicaciones

gráficas. En este curso, nos centraremos en el uso por medio de línea de comandos.

Los SCV, en su mayoría -y a rasgos generales- cuentan con un conjunto de funcionalidades, las cuales, para cada una, existe un determinado comando (generalmente, similar en la mayoría de los SCV).

Para **instalar Bazaar en tu ordenador**, ejecuta el siguiente comando:

```
sudo apt-get install bzr
```

Una vez instalado Bazaar, deberás **clonar el repositorio central** (desde el servidor del curso) a tu ordenador local:

```
bzr branch sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
```

A continuación, deberás ingresar tu contraseña.

Una vez clonado el repositorio, deberás agregar unas líneas al archivo de configuración de Bazaar. Para ello, abre el archivo de configuración con el editor Nano:

```
nano trunk/.bzr/branch/branch.conf
```

Mueve el cursor hasta la siguiente línea y pulsa las teclas Ctrl + K:

```
parent_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
```

A continuación, pulsa tres veces, las teclas Ctrl + U para pegar (tres veces) la línea que cortaste anteriormente. Deberás ver lo siguiente:

```
parent_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
parent_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
parent_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
```

Reemplaza la palabra "*parent*" de la segunda línea, por "*push*" y la de la tercera, por "*pull*" de forma tal que el archivo, se vea como sigue:

```
parent_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
push_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
pull_location = sftp://tu_usuario@66.228.52.93/home/tu_usuario/public/trunk
```

Para guardar el archivo pulsa las teclas Ctrl + O (enter) y para salir, pulsa Ctrl + X.

02 Estructura y elementos del lenguaje

Dentro de los **lenguajes informáticos**, Python, pertenece al grupo de los **lenguajes de programación** y puede ser clasificado como un **lenguaje interpretado**, **de alto nivel**, **multiplataforma**, de **tipado dinámico** y **multiparadigma**. A diferencia de la mayoría de los lenguajes de programación, **Python nos provee de reglas de estilos**, a fin de poder escribir código fuente más legible y de manera estandarizada. Estas reglas de estilo, son definidas a través de la ***Python Enhancement Proposal* Nº 8 (PEP 8)** , la cual iremos viendo a lo largo del curso.

GLOSARIO

Lenguaje informático: es un idioma artificial, utilizado por ordenadores, cuyo fin es transmitir información de algo a alguien. Los lenguajes informáticos, pueden clasificarse en: a) lenguajes de programación (Python, PHP, Pearl, C, etc.); b) lenguajes de especificación (UML); c) lenguajes de consulta (SQL); d) lenguajes de marcas (HTML, XML); e) lenguajes de transformación (XSLT); f) protocolos de comunicaciones (HTTP, FTP); entre otros.

Lenguaje de programación: es un lenguaje informático, diseñado para expresar órdenes e instrucciones precisas, que deben ser llevadas a cabo por una computadora. El mismo puede utilizarse para crear programas que controlen el comportamiento físico o lógico de un ordenador. Está compuesto por una serie de símbolos, reglas sintácticas y semánticas que definen la estructura del lenguaje.

Lenguajes de alto nivel: son aquellos cuya característica principal, consiste en una estructura sintáctica y semántica legible, acorde a las capacidades cognitivas humanas. A diferencia de los lenguajes de bajo nivel, son independientes de la arquitectura del hardware, motivo por el cual, asumen mayor portabilidad.

Lenguajes interpretados: a diferencia de los compilados, no requieren de un compilador para ser ejecutados sino de un intérprete. Un intérprete, actúa de manera casi idéntica a un compilador, con la salvedad de que ejecuta el programa directamente, sin necesidad de generar previamente un ejecutable. Ejemplo de lenguajes de programación interpretado son Python, PHP, Ruby, Lisp, entre otros.

Tipado dinámico: un lenguaje de tipado dinámico es aquel cuyas variables, no requieren ser definidas asignando su tipo de datos, sino que éste, se auto-asigna en tiempo de ejecución, según el valor declarado.

Multiplataforma: significa que puede ser interpretado en diversos Sistemas Operativos como GNU/Linux, Windows, Mac OS, Solaris, entre otros.

Multiparadigma: acepta diferentes paradigmas (técnicas) de programación, tales como la orientación a objetos, aspectos, la programación imperativa y funcional.

Código fuente: es un conjunto de instrucciones y órdenes lógicas, compuestos de algoritmos que se encuentran escritos en un determinado lenguaje de programación, las cuales deben ser interpretadas o compiladas, para permitir la ejecución del programa informático.

Elementos del Lenguaje

Como en la mayoría de los lenguajes de programación de alto nivel, en Python se compone de una serie de elementos que alimentan su estructura. Entre ellos, podremos encontrar los siguientes:

Variables

Una variable es un espacio para almacenar datos modificables, en la memoria de un ordenador. En Python, una variable se define con la sintaxis:

```
nombre_de_la_variable = valor_de_la_variable
```

Cada variable, tiene un nombre y un valor, el cual define a la vez, el tipo de datos de la variable.

Existe un tipo de “variable”, denominada **constante**, la cual se utiliza para definir valores fijos, que no requieran ser modificados.

PEP 8: variables
Utilizar nombres descriptivos y en minúsculas. Para nombres compuestos, separar las palabras por guiones bajos. Antes y después del signo =, debe haber uno (y solo un) espacio en blanco

Correcto: `mi_variable = 12`
Incorrecto: `MiVariable = 12 | mivariable = 12 | mi_variable=12 | mi_variable  =  12`

PEP 8: constantes
Utilizar nombres descriptivos y en mayúsculas separando palabras por guiones bajos.

Ejemplo: `MI_CONSTANTE = 12`

Para **imprimir un valor en pantalla**, en Python, se utiliza la palabra clave `print`:

```
mi_variable = 15
print mi_variable
```

Lo anterior, imprimirá el valor de la variable mi_variable en pantalla.

Tipos de datos

Una variable (o constante) puede contener valores de diversos tipos. Entre ellos:

Cadena de texto (string):

```
mi_cadena = "Hola Mundo!"

mi_cadena_multilinea = """
Esta es una cadena
de varias lineas
"""
```

Número entero:

```
edad = 35
```

Número entero octal:

```
edad = 043
```

Número entero hexadecimal:

```
edad = 0x23
```

Número real:

```
precio = 7435.28
```

Booleano (verdadero / Falso):

```
verdadero = True
falso = False
```

Existen además, otros tipos de datos más complejos, que veremos más adelante.

Operadores Aritméticos

Entre los operadores aritméticos que Python utiliza, podemos encontrar los siguientes:

Símbolo	Significado	Ejemplo	Resultado
+	Suma	a = 10 + 5	a es 15
-	Resta	a = 12 - 7	a es 5
-	Negación	a = -5	a es -5
*	Multiplicación	a = 7 * 5	a es 35
**	Exponente	a = 2 ** 3	a es 8
/	División	a = 12.5 / 2	a es 6.25
//	División entera	a = 12.5 / 2	a es 6.0
%	Módulo	a = 27 % 4	a es 3

PEP 8: operadores
Siempre colocar un espacio en blanco, antes y después de un operador

Un ejemplo sencillo con variables y operadores aritméticos:

```
monto_bruto = 175
tasa_interes = 12
monto_interes = monto_bruto * tasa_interes / 100
tasa_bonificacion = 5
importe_bonificacion = monto_bruto * tasa_bonificacion / 100
monto_neto = (monto_bruto - importe_bonificacion) + monto_interes
```

Comentarios

Un archivo, no solo puede contener código fuente. También puede incluir comentarios (notas que como programadores, indicamos en el código para poder comprenderlo mejor).

Los comentarios pueden ser de dos tipos: de una sola línea o multi-línea y se expresan de la siguiente manera:

```
# Esto es un comentario de una sola línea
mi_variable = 15

"""Y este es un comentario
de varias líneas"""
mi_variable = 15

mi_variable = 15  # Este comentario es de una línea también
```

En los comentarios, pueden incluirse palabras que nos ayuden a identificar además, el subtipo de comentario:

```
# TODO esto es algo por hacer
# FIXME esto es algo que debe corregirse
# XXX esto también, es algo que debe corregirse
```

PEP 8: comentarios
Comentarios en la misma línea del código deben separarse con dos espacios en blanco. Luego del símbolo # debe ir un solo espacio en blanco.

Correcto:
```
a = 15  # Edad de María
```
Incorrecto:
```
a = 15 # Edad de María
```

Tipos de datos complejos

Python, posee además de los tipos ya vistos, 3 tipos más complejos, que admiten una **colección de datos**. Estos tipos son:

- Tuplas
- Listas
- Diccionarios

Estos tres tipos, pueden almacenar colecciones de datos de diversos tipos y se diferencian por su sintaxis y por la forma en la cual los datos pueden ser manipulados.

Tuplas

Una tupla **es una variable que permite almacenar varios datos inmutables** (no pueden ser modificados una vez creados) de tipos diferentes:

```
mi_tupla = ('cadena de texto', 15, 2.8, 'otro dato', 25)
```

Se puede acceder a cada uno de los datos mediante su índice correspondiente, siendo 0 (cero), el índice del primer elemento:

```
print mi_tupla[1]  # Salida: 15
```

También se puede acceder a una porción de la tupla, indicando (opcionalmente) desde el índice de inicio hasta el índice de fin:

```
print mi_tupla[1:4]  # Devuelve: (15, 2.8, 'otro dato')
print mi_tupla[3:]   # Devuelve: ('otro dato', 25)
print mi_tupla[:2]   # Devuelve: ('cadena de texto', 15)
```

Otra forma de acceder a la tupla de forma inversa (de atrás hacia adelante), es colocando un índice negativo:

```
print mi_tupla[-1]  # Salida: 25
print mi_tupla[-2]  # Salida: otro dato
```

Listas

Una lista es similar a una tupla con la diferencia fundamental de que permite modificar los datos una vez creados

```
mi_lista = ['cadena de texto', 15, 2.8, 'otro dato', 25]
```

A las listas se accede igual que a las tuplas, por su número de índice:

```
print mi_lista[1]    # Salida: 15
print mi_lista[1:4]  # Devuelve: [15, 2.8, 'otro dato']
print mi_lista[-2]   # Salida: otro dato
```

Las lista NO son inmutables: permiten modificar los datos una vez creados:

```
mi_lista[2] = 3.8  # el tercer elemento ahora es 3.8
```

Las listas, a diferencia de las tuplas, permiten agregar nuevos valores:

```
mi_lista.append('Nuevo Dato')
```

Diccionarios

Mientras que a las listas y tuplas se accede solo y únicamente por un número de índice, los diccionarios permiten utilizar una clave para declarar y acceder a un valor:

```
mi_diccionario = {'clave_1': valor_1, 'clave_2': valor_2, \
                  'clave_7': valor_7}
print mi_diccionario['clave_2']  # Salida: valor_2
```

Un diccionario permite eliminar cualquier entrada:

```
del(mi_diccionario['clave_2'])
```

Al igual que las listas, el diccionario permite modificar los valores

```
mi_diccionario['clave_1'] = 'Nuevo Valor'
```

Estructuras de Control de Flujo

Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada. En este capítulo, hablaremos sobre dos estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas

Identación

Para hablar de estructuras de control de flujo en Python, es imprescindible primero, hablar de identación.

¿Qué es la identación? En un lenguaje informático, la identación es lo que la sangría al lenguaje humano escrito (a nivel formal). Así como para el lenguaje formal, cuando uno redacta una carta, debe respetar ciertas sangrías, los lenguajes informáticos, requieren una identación.

No todos los lenguajes de programación, necesitan de una identación, aunque sí, se estila implementarla, a fin de otorgar mayor legibilidad al código fuente. Pero **en el caso de Python, la identación es obligatoria**, ya que de ella, dependerá su estructura.

> **PEP 8: identación**
> Una identación de **4 (cuatro) espacios en blanco**, indicará que las instrucciones identadas, forman parte de una misma estructura de control.

Una estructura de control, entonces, se define de la siguiente forma:

```
inicio de la estructura de control:
    expresiones
```

Encoding

El **encoding** (o codificación) es otro de los elementos del lenguaje que no puede omitirse a la hora de hablar de estructuras de control.

> El **encoding** no es más que una **directiva que se coloca al inicio de un archivo Python, a fin de indicar al sistema, la codificación de caracteres utilizada en el archivo.**

```
# -*- coding: utf-8 -*-
```

utf-8 podría ser cualquier codificación de caracteres. Si no se indica una codificación de caracteres, Python podría producir un error si encontrara caracteres "extraños":

```
print "En el Ñágara encontré un Ñandú"
```

Producirá un error de sintaxis: `SyntaxError: Non-ASCII character[...]`

En cambio, indicando el encoding correspondiente, el archivo se ejecutará con éxito:

```
# -*- coding: utf-8 -*-

print "En el Ñágara encontré un Ñandú"
```

Produciendo la siguiente salida:

```
En el Ñágara encontré un Ñandú
```

Asignación múltiple

Otra de las ventajas que Python nos provee, es la de poder asignar en una sola instrucción, múltiples variables:

```
a, b, c = 'string', 15, True
```

En una sola instrucción, estamos declarando tres variables: a, b y c y asignándoles un valor concreto a cada una:

```
>>> print a
string
>>> print b
15
>>> print c
True
```

La asignación múltiple de variables, también puede darse utilizando como valores, el

contenido de una tupla:

```
>>> mi_tupla = ('hola mundo', 2011)
>>> texto, anio = mi_tupla
>>> print texto
hola mundo
>>> print anio
2011
```

O también, de una lista:

```
>>> mi_lista = ['Argentina', 'Buenos Aires']
>>> pais, provincia = mi_lista
>>> print pais
Argentina
>>> print provincia
Buenos Aires
```

Estructuras de control de flujo condicionales

> *“[...] Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición [...]”*
>
> *Cita textual del libro “Python para Todos” de Raúl González Duque (http://mundogeek.net/tutorial-python/)*

Las estructuras de control condicionales, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar. **La evaluación de condiciones**, solo **puede arrojar** 1 de 2 resultados: **verdadero o falso** (True o False).

En la vida diaria, actuamos de acuerdo a la evaluación de condiciones, de manera mucho más frecuente de lo que en realidad creemos: ***Si** el semáforo está en verde, cruzar la calle. **Sino**, esperar a que el semáforo se ponga en verde.* A veces, también evaluamos más de una condición para ejecutar una determinada acción: *Si llega la factura de la luz **y** tengo dinero, pagar la boleta.*

Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación):

```
 OPERADORES RELACIONALES (DE COMPARACIÓN)
----------------------------------------------------------
 Símbolo   Significado          Ejemplo         Resultado
 ==        Igual que            5 == 7          Falso
 !=        Distinto que         rojo != verde   Verdadero
 <         Menor que            8 < 12          Verdadero
 >         Mayor que            12 > 7          Falso
 <=        Menor o igual que    12 <= 12        Verdadero
 >=        Mayor o igual que    4 >= 5          Falso
```

Y para evaluar más de una condición simultáneamente, se utilizan **operadores lógicos**:

```
 OPERADORES LÓGICOS
----------------------------------------------------------
 Operador          Ejemplo              Resultado*
 and (y)           5 == 7 and 7 < 12    0 y 0         Falso
                   9 < 12 and 12 > 7    1 y 1         Verdadero
                   9 < 12 and 12 > 15   1 y 0         Falso
 or (o)            12 == 12 or 15 < 7   1 o 0         Verdadero
                   7 > 5 or 9 < 12      1 o 1         Verdadero
 xor               4 == 4 xor 9 > 3     1 o 1         Falso
 (o excluyente)    4 == 4 xor 9 < 3     1 o 0         Verdadero
```

(*) 1 indica resultado verdadero de la condición, mientras que 0, indica falso.

Las estructuras de control de flujo condicionales, se definen mediante el uso de tres palabras claves reservadas, del lenguaje: **`if`** (si), **`elif`** (sino, si) y **`else`** (sino).

Veamos algunos ejemplos:

> Si semáforo esta en verde, cruzar la calle. Sino, esperar.

```
if semaforo == verde:
    print "Cruzar la calle"
else:
    print "Esperar"
```

> Si gasto hasta $100, pago con dinero en efectivo. Sino, si gasto más de $100 pero menos de $300, pago con tarjeta de débito. Sino, pago con tarjeta de crédito.

```
if compra <= 100:
    print "Pago en efectivo"
elif compra > 100 and compra < 300:
    print "Pago con tarjeta de débito"
else:
    print "Pago con tarjeta de crédito"
```

> Si la compra es mayor a $100, obtengo un descuento del 10%

```
importe_a_pagar = total_compra

if total_compra > 100:
    tasa_descuento = 10
    importe_descuento = total_compra * tasa_descuento / 100
    importe_a_pagar = total_compra – importe_descuento
```

Estructuras de control iterativas

A diferencia de las estructuras de control condicionales, las iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

En Python se dispone de dos estructuras cíclicas:

- El bucle **while**
- El bucle **for**

Las veremos en detalle a continuación.

Bucle while

Este bucle, se encarga de ejecutar una misma acción "mientras que" una determinada condición se cumpla:

> Mientras que año sea menor o igual a 2012, imprimir la frase "Informes del Año *año*"

```
# -*- coding: utf-8 -*-

anio = 2001
while anio <= 2012:
    print "Informes del Año", str(anio)
    anio += 1
```

La iteración anterior, generará la siguiente salida:

```
Informes del año 2001
Informes del año 2002
Informes del año 2003
Informes del año 2004
Informes del año 2005
Informes del año 2006
Informes del año 2007
Informes del año 2008
Informes del año 2009
Informes del año 2010
Informes del año 2011
Informes del año 2012
```

Si miras la última línea:

```
anio += 1
```

Podrás notar que en cada iteración, incrementamos el valor de la variable que condiciona el bucle (`anio`). Si no lo hiciéramos, esta variable siempre sería igual a `2001` y el bucle se ejecutaría de forma infinita, ya que la condición (`anio <= 2012`) siempre se estaría cumpliendo.

Pero ¿Qué sucede si el valor que condiciona la iteración no es numérico y no puede incrementarse? En ese caso, podremos utilizar una estructura de control condicional, anidada dentro del bucle, y frenar la ejecución cuando el condicional deje de cumplirse, con la palabra clave reservada break:

```
while True:
    nombre = raw_input("Indique su nombre: ")
    if nombre:
        break
```

El bucle anterior, incluye un condicional anidado que verifica si la variable nombre es verdadera (solo será verdadera si el usuario tipea un texto en pantalla cuando el nombre le es solicitado). Si es verdadera, el bucle para (break). Sino, seguirá ejecutándose hasta que el usuario, ingrese un texto en pantalla.

Bucle for

El bucle for, en Python, es aquel que nos permitirá iterar sobre una variable compleja, del tipo lista o tupla:

> Por cada nombre en mi_lista, imprimir nombre

```
mi_lista = ['Juan', 'Antonio', 'Pedro', 'Herminio']
for nombre in mi_lista:
    print nombre
```

> Por cada color en mi_tupla, imprimir color

```
mi_tupla = ('rosa', 'verde', 'celeste', 'amarillo')
for color in mi_tupla:
    print color
```

En los ejemplos anteriores, nombre y color, son dos variables declaradas en tiempo de ejecución (es decir, se declaran dinámicamente durante el bucle), asumiendo como valor, el de cada elemento de la lista (o tupla) en cada iteración.

Otra forma de iterar con el bucle for, puede emular a while:

> Por cada año en el rango 2001 a 2013, imprimir la frase “Informes del Año *año*”

```
# -*- coding: utf-8 -*-
for anio in range(2001, 2013):
    print "Informes del Año", str(anio)
```

03 Módulos, paquetes y namespaces

En Python, cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado **`__init__.py`**. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

Creando módulos empaquetados

En Python, cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado **`__init__.py`**. Este archivo, no necesita contener ninguna instrucción. De hecho, puede estar completamente vacío.

```
.
└── paquete
    ├── __init__.py
    ├── modulo1.py
    ├── modulo2.py
    └── modulo3.py
```

Los paquetes, a la vez, también pueden contener otros sub-paquetes:

```
.
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Y los módulos, no necesariamente, deben pertenecer a un paquete:

```
.
├── modulo1.py
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Importando módulos enteros

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario **importar los módulos** que se quieran utilizar. Para importar un módulo, se utiliza la instrucción **`import`**, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el .py) que se desee importar.

```
# -*- coding: utf-8 *-*

import modulo  # importar un módulo que no pertenece a un paquete
import paquete.modulo1  # importar un módulo que está dentro de un paquete
import paquete.subpaquete.modulo1
```

La instrucción `import` seguida de `nombre_del_paquete.nombre_del_modulo`, nos permitirá hacer uso de todo el código que dicho módulo contenga.

> **Python tiene sus propios módulos**, los cuales forman parte de su **librería de módulos estándar**, que también pueden ser importados.

Namespaces

Para **acceder** (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el ***namespace***, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un namespace, es el nombre que se ha indicado luego de la palabra `import`, es decir la ruta (namespace) del módulo:

```
print modulo.CONSTANTE_1
print paquete.modulo1.CONSTANTE_1
print paquete.subpaquete.modulo1.CONSTANTE_1
```

Alias

Es posible también, abreviar los namespaces mediante un "alias". Para ello, durante la importación, se asigna la palabra clave **`as`** seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

```
import modulo as m
import paquete.modulo1 as pm
import paquete.subpaquete.modulo1 as psm
```

Luego, para acceder a cualquier elemento de los módulos importados, el namespace utilizado será el alias indicado durante la importación:

```
print m.CONSTANTE _1
print pm.CONSTANTE _1
print psm.CONSTANTE_1
```

Importar módulos sin utilizar namespaces

En Python, es posible también, importar de un módulo solo los elementos que se desee utilizar. Para ello se utiliza la instrucción `from` seguida del namespace, más la instrucción `import` seguida del elemento que se desee importar:

```
from paquete.modulo1 import CONSTANTE_1
```

En este caso, se accederá directamente al elemento, sin recurrir a su namespace:

```
print CONSTANTE_1
```

Es posible también, importar más de un elemento en la misma instrucción. Para ello, cada elemento irá separado por una coma (,) y un espacio en blanco:

```
from paquete.modulo1 import CONSTANTE_1, CONSTANTE_2
```

Pero ***¿qué sucede si los elementos importados desde módulos diferentes tienen los mismos nombres?*** En estos casos, habrá que **prevenir fallos, utilizando alias para los elementos**:

```
from paquete.modulo1 import CONSTANTE_1 as C1, CONSTANTE_2 as C2
from paquete.subpaquete.modulo1 import CONSTANTE_1 as CS1, CONSTANTE_2 as CS2

print C1
print C2
print CS1
print CS2
```

PEP 8: importación
La importación de módulos debe realizarse al comienzo del documento, en orden alfabético de paquetes y módulos.
Primero deben importarse los módulos propios de Python. Luego, los módulos de terceros y finalmente, los módulos propios de la aplicación.
Entre cada bloque de imports, debe dejarse una línea en blanco.

De forma alternativa (pero muy poco recomendada), también es posible importar todos los elementos de un módulo, sin utilizar su namespace pero tampoco alias. Es decir, que todos los elementos importados se accederá con su nombre original:

```
from paquete.modulo1 import *

print CONSTANTE_1
print CONSTANTE_2
```

#TODO: Abrir una terminal e iniciar el shell interactivo (intérprete) de Python. A continuación, importar el módulo `this`:
`import this`

04 Funciones definidas por el usuario

Una función, es la forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas. Es decir, que al colocar un algoritmo dentro de una función, al correr el archivo, el algoritmo no será ejecutado si no se ha hecho una referencia a la función que lo contiene.

Definiendo funciones

En Python, la definición de funciones se realiza mediante la instrucción `def` más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre. Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el algoritmo que la compone, irá identado con 4 espacios:

```
def mi_funcion():
    # aquí el algoritmo
```

Una función, no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print "Hola Mundo"

funcion()
```

Cuando una función, haga un **retorno de datos**, éstos, pueden ser asignados a una variable:

```
def funcion():
    return "Hola Mundo"

frase = funcion()
print frase
```

Sobre los parámetros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
def mi_funcion(nombre, apellido):
    # algoritmo
```

> *Los parámetros, **se indican** entre los paréntesis, **a modo de variables**, a fin de poder utilizarlos como tales, dentro de la misma función.*

Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de **variables de ámbito local**. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print nombre_completo
```

Si quisiéramos acceder a esas variables locales, fuera de la función, obtendríamos un error:

```
def mi_funcion(nombre, apellido):
    nombre_completo = nombre, apellido
    print nombre_completo

print nombre  # Retornará el error: NameError: name 'nombre' is not defined
```

Al llamar a una función, ***siempre se le deben pasar sus argumentos en el mismo orden en el que los espera****. Pero esto* ***puede evitarse****, haciendo uso del paso de argumentos como keywords (ver más abajo: "****Keywords como parámetros****").*

Parámetros por omisión

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre

saludar('Pepe Grillo')  # Imprime: Hola Pepe Grillo
```

PEP 8: Funciones
A la definición de una función la deben anteceder dos líneas en blanco.
Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo =.

Keywords como parámetros

En Python, también es posible llamar a una función, pasándole los argumentos esperados, como pares de `claves=valor`:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre
```

```
saludar(mensaje="Buen día", nombre="Juancho")
```

Parámetros arbitrarios

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de tupla.

Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco (*):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios):
    print parametro_fijo

    # Los parámetros arbitrarios se corren como tuplas
    for argumento in arbitrarios:
        print argumento

recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario 2',
                                'arbitrario 3')
```

> *Si una función espera recibir parámetros fijos y arbitrarios, **los arbitrarios siempre deben suceder a los fijos**.*

Es posible también, obtener parámetros arbitrarios como pares de clave=valor. En estos casos, al nombre del parámetro deben precederlo dos astericos (**):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios, **kwords):
    print parametro_fijo
    for argumento in arbitrarios:
        print argumento

    # Los argumentos arbitrarios tipo clave, se recorren como los diccionarios
    for clave in kwords:
        print "El valor de", clave, "es", kwords[clave]

recorrer_parametros_arbitrarios("Fixed", "arbitrario 1", "arbitrario 2",
                                "arbitrario 3", clave1="valor uno",
                                clave2="valor dos")
```

Desempaquetado de parámetros

Puede ocurrir además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla. En este caso, el signo asterisco (*) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función:

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [1500, 10]
print calcular(*datos)
```

El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (**):

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = {"descuento": 10, "importe": 1500}
print calcular(**datos)
```

Llamadas de retorno

En Python, es posible (al igual que en la gran mayoría de los lenguajes de programación), llamar a una función dentro de otra, de forma fija y de la misma manera que se la llamaría, desde fuera de dicha función:

```
def funcion():
    return "Hola Mundo"

def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre
    print mi_funcion()
```

Sin embargo, es posible que se desee **realizar dicha llamada, de manera dinámica**, es decir, **desconociendo el nombre de la función** a la que se deseará llamar. A este tipo de acciones, se las denomina **llamadas de retorno**.

Para conseguir llamar a una función de manera dinámica, Python dispone de dos funciones nativas: **`locals()`** y **`globals()`**

Ambas funciones, retornan un diccionario. En el caso de `locals()`, éste diccionario se compone -justamente- de todos los elementos de ámbito local, mientras que el de `globals()`, retorna lo propio pero a nivel global.

```
def funcion():
    return "Hola Mundo"


def llamada_de_retorno(func=""):
    """Llamada de retorno a nivel global"""
    return globals()[func]()
```

```
print llamada_de_retorno("funcion")

# Llamada de retorno a nivel local
nombre_de_la_funcion = "funcion"
print locals()[nombre_de_la_funcion]()
```

Si se tienen que pasar argumentos en una llamada retorno, se lo puede hacer normalmente:

```
def funcion(nombre):
    return "Hola " + nombre


def llamada_de_retorno(func=""):
    """Llamada de retorno a nivel global"""
    return globals()[func]("Laura")

print llamada_de_retorno("funcion")

# Llamada de retorno a nivel local
nombre_de_la_funcion = "funcion"
print locals()[nombre_de_la_funcion]("Facundo")
```

Saber si una función existe y puede ser llamada

Durante una llamada de retorno, el nombre de la función, puede no ser el indicado. Entonces, siempre que se deba realizar una llamada de retorno, es necesario comprobar que ésta exista y pueda ser llamada.

```
if nombre_de_la_funcion in locals():
    if callable(locals()[nombre_de_la_funcion]):
        print locals()[nombre_de_la_funcion]("Emilse")
```

El operador `in`, nos permitirá conocer si un elemento se encuentra dentro de una colección, mientras que la función `callable()` nos dejará saber si esa función puede ser llamada.

```
def funcion(nombre):
    return "Hola " + nombre


def llamada_de_retorno(func=""):
    if func in globals():
        if callable(globals()[func]):
            return globals()[func]("Laura")
    else:
        return "Función no encontrada"

print llamada_de_retorno("funcion")


nombre_de_la_funcion = "funcion"
```

```
if nombre_de_la_funcion in locals():
    if callable(locals()[nombre_de_la_funcion]):
        print locals()[nombre_de_la_funcion]("Facundo")
else:
    print "Función no encontrada"
```

Llamadas recursivas

Se denomina llamada recursiva (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia sí misma.

Las llamadas recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas preventivas adecuadas y, solo utilizarse cuando sea estrictamente necesario y no exista una forma alternativa viable, que resuelva el problema evitando la recursividad.

Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

```
def jugar(intento=1):
    respuesta = raw_input("¿De qué color es una naranja? ")
    if respuesta != "naranja":
        if intento < 3:
            print "\nFallaste! Inténtalo de nuevo"
            intento += 1
            jugar(intento)  # Llamada recursiva
        else:
            print "\nPerdiste!"
    else:
        print "\nGanaste!"

jugar()
```

Sobre la finalidad de las funciones

Una función, puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos y, utilizar cualquiera de las características vistas hasta ahora. No obstante ello, **una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción, reutilizable y por lo tanto, tan genérica como sea posible**.

05 Introducción a la Orientación a Objetos

En Python todo es un “objeto” y debe ser manipulado -y entendido- como tal. Pero ¿Qué es un objeto? ¿De qué hablamos cuando nos referimos a “orientación a objetos? En este capítulo, haremos una introducción que responderá a estas -y muchas otras- preguntas.

Nos enfocaremos primero, en cuestiones de conceptos básicos, para luego, ir introduciéndonos de a poco, en principios teóricos elementalmente necesarios, para implementar la orientación a objetos en la práctica.

Pensar en objetos

Pensar en objetos, puede resultar -al inicio- una tarea difícil. Sin embargo, difícil no significa complejo. Por el contrario, pensar en objetos representa la mayor simplicidad que uno podría esperar del mundo de la programación. **Pensar en objetos, es simple**... aunque lo simple, no necesariamente signifique sencillo.

Y ¿qué es un objeto?

Pues, como dije antes, es "simple". Olvidemos los formalismos, la informática y todo lo que nos rodea. Simplemente, olvida todo y concéntrate en lo que sigue. Lo explicaré de manera "simple":

Un objeto es "una cosa". Y, si una cosa es un sustantivo, entonces **un objeto es un sustantivo**.

Mira a tu alrededor y encontrarás decenas, cientos de objetos. Tu ordenador, es un objeto. Tú, eres un objeto. Tu llave es un objeto. El cenicero (ese que tienes frente a ti cargado de colillas de cigarrillo), es otro objeto. Tu mascota también es un objeto.

> *Cuando pensamos en "objetos", **todos los sustantivos son objetos**.*

Sencillo ¿cierto? Entonces, de ahora en más, solo concéntrate en pensar la vida en objetos (al menos, hasta terminar de leer este documento).

Ahora ¿qué me dices si describimos las cualidades de un objeto?

Describir un objeto, es simplemente mencionar sus cualidades. **Las cualidades son adjetivos**. Si no sabes que es un adjetivo, estamos jodidos (y mucho). Pero, podemos decir que **un adjetivo es una cualidad del sustantivo**.

Entonces, para describir "la manera de ser" de un objeto, debemos preguntarnos **¿cómo es el objeto?** Toda respuesta que comience por "el objeto es", seguida de un adjetivo, será una cualidad del objeto.

Algunos ejemplos:

– **El objeto es** verde

– **El objeto es** grande

- **El objeto es** feo

Ahora, imagina que te encuentras frente a un niño de 2 años (niño: objeto que pregunta cosas que tú das por entendidas de forma implícita). Y cada vez que le dices las cualidades de un objeto al ~~molesto~~ niño-objeto, éste te pregunta: -"¿Qué es...?", seguido del adjetivo con el cuál finalizaste tu frase. Entonces, tu le respondes diciendo "es un/una" seguido de un sustantivo. Te lo muestro con un ejemplo:

- El objeto es verde. **¿Qué es** verde? Un color.
- El objeto es grande. **¿Qué es** grande? Un tamaño.
- El objeto es feo. **¿Qué es** feo? Un aspecto.

Estos sustantivos que responden a la pregunta del niño, pueden pasar a formar parte de una **locución adjetiva** que especifique con mayor precisión, las descripciones anteriores:

- El objeto es **de *color*** verde.
- El objeto es **de *tamaño*** grande.
- El objeto es **de *aspecto*** feo.

Podemos decir entonces -y todo esto, gracias al ~~molesto~~ niño-objeto-, que una cualidad, es un atributo (derivado de "cualidad *atribuible* a un objeto") y que entonces, **un objeto es un sustantivo que posee atributos, cuyas cualidades lo describen.**

Veámoslo más gráficamente:

OBJETO (sustantivo)	**ATRIBUTO (locución adjetiva)**	**CUALIDAD DEL ATRIBUTO (adjetivo)**
(el) Objeto	(es de) color	Verde
	(es de) tamaño	Grande
	(es de) aspecto	Feo

Pero algunos objetos, también se componen de otros objetos...

Además de cualidades (locución adjetiva seguida de un adjetivo), **los objetos "tienen otras cosas"**. Estas "otras cosas", son aquellas "pseudo-cualidades" que en vez de responder a ¿cómo es el objeto? responden a **"¿cómo está compuesto el objeto?"** o incluso, aún más simple **"¿Qué tiene el objeto?".**

La respuesta a esta pregunta, estará dada por la frase "el objeto tiene...", seguida de un adverbio de cantidad (uno, varios, muchos, algunos, unas cuantas) y un sustantivo.

Algunos ejemplos:

– El objeto **tiene *algunas*** antenas
– El objeto **tiene *un*** ojo
– El objeto **tiene *unos cuantos*** pelos

Los componentes de un objeto, también integran los atributos de ese objeto. Solo que **estos atributos**, son algo particulares: **son otros objetos que poseen sus propias cualidades**. Es decir, que estos “atributos-objeto” también responderán a la pregunta “¿Cómo es/son ese/esos/esas?” seguido del atributo-objeto (sustantivo).

Ampliemos el ejemplo para que se entienda mejor:

- El objeto tiene algunas antenas. ¿***Cómo son esas*** antenas?
    - Las antenas son de color violeta
    - Las antenas son de longitud extensa
- El objeto tiene un ojo. ¿***Cómo es ese*** ojo?
    - El ojo es de forma oval
    - El ojo es de color azul
    - El ojo es de tamaño grande
- El objeto tiene unos cuantos pelos. ¿***Cómo son esos*** pelos?
    - Los pelos son de color fucsia
    - Los pelos son de textura rugosa

Pongámoslo más gráfico:

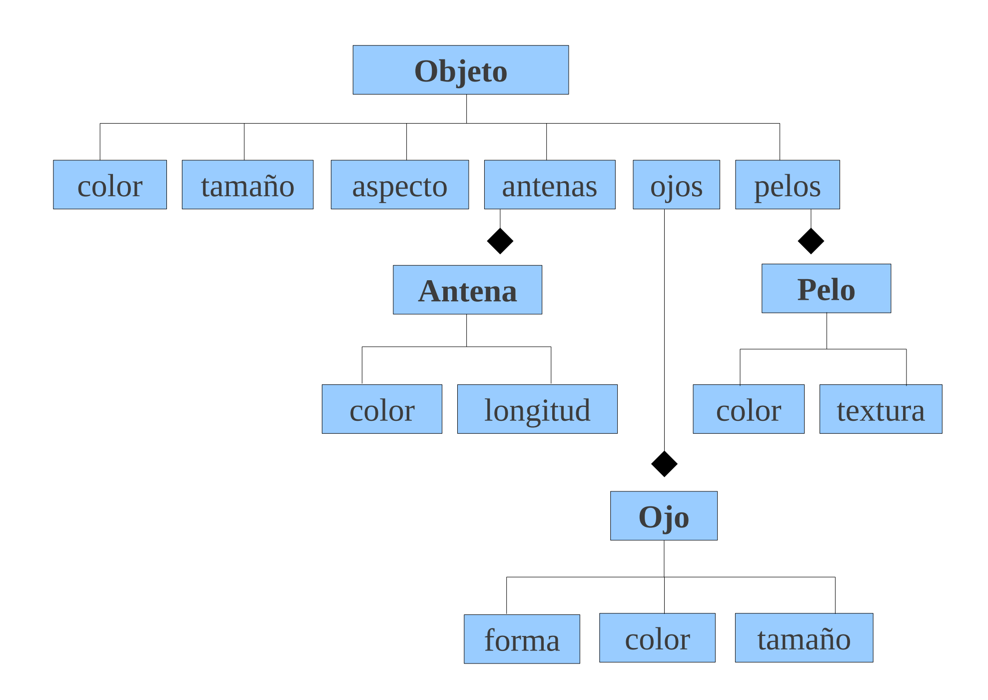
OBJETO (sustantivo)	**ATRIBUTO-OBJETO (sustantivo)**	**ATRIBUTOS (locución adjetiva)**	**CUALIDADES DE LOS ATRIBUTOS (adjetivo)**
(el) Objeto	(tiene algunas) antenas	(de) color (de) longitud	Violeta extensa
	(tiene un) ojo	(de) forma (de) color (de) tamaño	Oval azul grande
	(tiene unos cuantos) pelos	(de) color (de) textura	Fucsia rugosa

Entonces, podemos deducir que **un objeto puede tener dos tipos de atributos**:

1) Los que responden a la pregunta ***“¿Cómo es el objeto?”*** con la frase ***“El objeto es...”*** + adjetivo (atributos definidos por cualidades)
2) Los que responden a la pregunta ***“¿Qué tiene el objeto?”*** con la frase ***“El objeto***

tiene..." + sustantivo (atributos definidos por las cualidades de otro objeto)

Veámoslo aún, más gráficamente:



Viendo el gráfico anterior, tenemos lo siguiente: *Un **objeto** (sustantivo) al cual hemos descrito con tres **atributos** (adjetivos) y otros tres **atributos-objeto** (sustantivos) los cuáles son a la vez, otros tres objetos (sustantivos) con sus atributos (adjetivos) correspondientes. ¿Simple, no? Ahora, compliquemos todo un poco.*

Y también hay objetos que comparten características con otros objetos

*Resulta ser, que nuestro Objeto, es prácticamente igual a un nuevo objeto. Es decir, que el nuevo objeto que estamos viendo, tiene absolutamente todas las características que nuestro primer objeto, es decir, tiene los mismos atributos. Pero también, tiene algunas más. Por ejemplo, este **nuevo objeto**, además de los atributos de nuestro primer objeto, **tiene un pie**. Es decir, que las características de nuestro nuevo objeto, serán todas las del objeto original, más una nueva: pie.*

Repasemos las características de nuestro nuevo objeto:

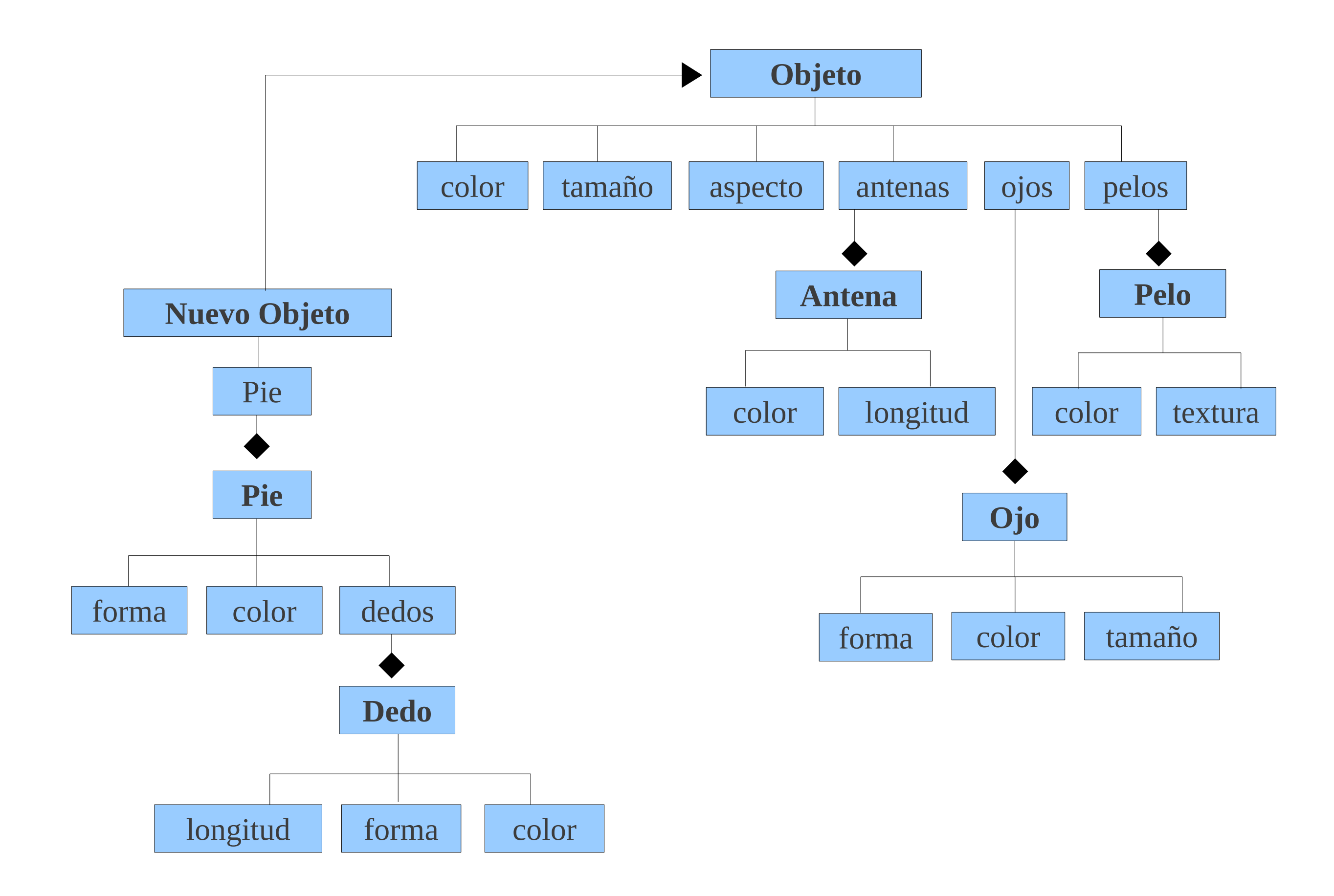
- El nuevo objeto es de color verde.
- El nuevo objeto es de tamaño grande.
- El nuevo objeto es de aspecto feo.

- El nuevo objeto tiene algunas antenas. ¿Cómo son esas antenas?
  - Las antenas son de color violeta
  - Las antenas son de longitud extensa
- El nuevo objeto tiene un ojo. ¿Cómo es ese ojo?
  - El ojo es de forma oval
  - El ojo es de color azul
  - El ojo es de tamaño grande
- El nuevo objeto tiene unos cuantos pelos. ¿Cómo son esos pelos?
  - Los pelos son de color fucsia
  - Los pelos son de textura rugosa

(nuevas características)

- El nuevo objeto tiene un pie. ¿Cómo es ese pie?
  - El pie es de forma rectangular
  - El pie es de color amarillo
  - El pie tiene 3 dedos. ¿Cómo son esos dedos?
    - Los dedos son de longitud mediana
    - Los dedos son de forma alargada
    - Los dedos son de color amarillo

Veamos todas las características de este nuevo, en un gráfico como lo hicimos antes.



Con mucha facilidad, podemos observar como nuestro nuevo objeto es una especie de "objeto original ampliado". Es decir que el nuevo objeto, es exactamente igual al objeto original (comparte todos sus atributos) pero posee nuevas características.

Está claro además, que el objeto original y el nuevo objeto, son dos objetos diferentes ¿cierto? No obstante, **el nuevo objeto es un sub-tipo del objeto original**.

Ahora sí, a complicarnos aún más.

Los objetos, también tienen la capacidad de "hacer cosas"

Ya describimos las cualidades de nuestros objetos. Pero de lo que no hemos hablado, es de aquellas cosas que los objetos "pueden hacer", es decir, "cuáles son sus capacidades".

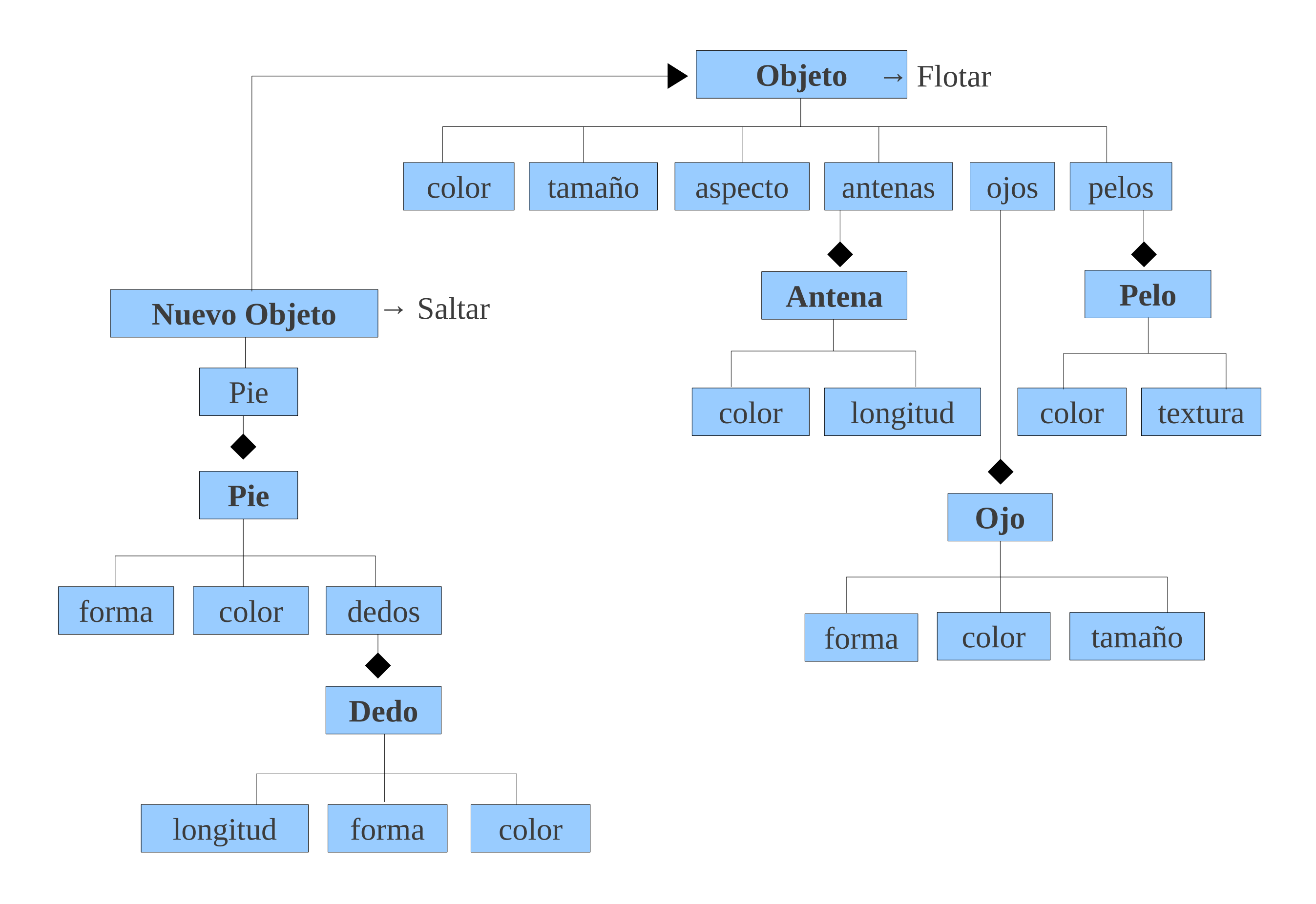
Los objetos tiene la capacidad de realizar acciones. Las acciones, son verbos. Es decir, que para conocer las capacidades de un objeto, debes preguntarte **"¿Qué puede hacer el objeto?"** y la respuesta a esta pregunta, estará dada por todas aquellas que comiencen por la frase "el objeto puede" seguida de un verbo en infinitivo.

Algunos ejemplos:

– El objeto original **puede** flotar

– El nuevo objeto (además) **puede** saltar

Si completamos el gráfico anterior con las acciones, obtendremos lo siguiente:



Si observas el gráfico anterior, notarás que el nuevo objeto, no solo tiene los mismos atributos que el objeto original, sino que además, también puede realizar las mismas acciones que éste. Sencillo, cierto?

Ahora sí, compliquémonos del todo :)

Objetos y más objetos: la parte difícil

Si entendiste todo lo anterior, ahora viene la parte difícil. ¿Viste que esto de "pensando en objetos" viene a colación de la programación orientada a objetos? Bueno, la parte difícil es que en la programación, todo lo que acabamos de ver, se denomina de una forma particular. Pero, la explicación es la misma que te di antes.

Al pan, pan. Y al vino, vino. Las cosas por su nombre

Cuando en el documento...	En la programación se denomina...	Y con respecto a la programación orientada a objetos es...
Hablamos de "objeto"	**Objeto**	Un elemento
Hablamos de "atributos" (o cualidades)	**Propiedades**	Un elemento
Hablamos de "acciones" que puede realizar el objeto	**Métodos**	Un elemento
Hablamos de "atributos-objeto"	**Composición**	Una técnica
Vemos que los objetos relacionados entre sí, tienen nombres de atributos iguales (por ejemplo: color y tamaño) y sin embargo, pueden tener valores diferentes	**Polimorfismo**	Una característica
Hablamos de objetos que son sub-tipos (o ampliación) de otros	**Herencia**	Una característica

Ahora, pasemos a un marco un poco más "académico".

Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP por sus siglas en inglés), es un **paradigma de programación**.

> ***Paradigma*: teoría cuyo núcleo central** *[...]* **suministra la base y modelo para resolver problemas** *[...] Definición de la Real Academia Española, vigésimo tercera edición*

Cómo tal, nos enseña un método -probado y estudiado- el cual se basa en las interacciones de objetos (todo lo descrito en el título anterior, “Pensar en objetos”) para resolver las necesidades de un sistema informático.

Básicamente, este paradigma se compone de 6 elementos y 7 características que veremos a continuación.

Elementos y Características de la POO

Los **elementos** de la POO, pueden entenderse como los ***“materiales”*** que necesitamos para diseñar y programar un sistema, mientras que las **características**, podrían asumirse como las ***“herramientas”*** de las cuáles disponemos para construir el sistema con esos materiales.

Entre los **elementos principales** de la POO, podremos encontrar a:

Clases

Las clases son los modelos sobre los cuáles se construirán nuestros objetos. Podemos tomar como ejemplo de clases, el gráfico que hicimos en la página 8 de este documento.

En Python, una clase se define con la instrucción `class` seguida de un nombre genérico para el objeto.

```
class Objeto:
    pass


class Antena:
    pass


class Pelo:
```

```
    pass


class Ojo:
    pass
```

> **PEP 8: clases**
> El nombre de las clases se define en singular, utilizando CamelCase.

Propiedades

Las propiedades, como hemos visto antes, son las características intrínsecas del objeto. Éstas, se representan a modo de variables, solo que técnicamente, pasan a denominarse "propiedades":

```
class Antena():
    color = ""
    longitud = ""


class Pelo():
    color = ""
    textura = ""


class Ojo():
    forma = ""
    color = ""
    tamanio = ""


class Objeto():
    color = ""
    tamanio = ""
    aspecto = ""
    antenas = Antena()  # propiedad compuesta por el objeto objeto Antena
    ojos = Ojo()        # propiedad compuesta por el objeto objeto Ojo
    pelos = Pelo()      # propiedad compuesta por el objeto objeto Pelo
```

> **PEP 8: propiedades**
> Las propiedades se definen de la misma forma que las variables (aplican las mismas reglas de estilo).

Métodos

Los métodos son "funciones" (como las que vimos en el capítulo anterior), solo que

técnicamente se denominan métodos, y representan acciones propias que puede realizar el objeto (y no otro):

```
class Objeto():
    color = "verde"
    tamanio = "grande"
    aspecto = "feo"
    antenas = Antena()
    ojos = Ojo()
    pelos = Pelo()


    def flotar(self):
        pass
```

> *Notar que el primer parámetro de un método, siempre debe ser* `self`.

Objeto

Las clases por sí mismas, no son más que modelos que nos servirán para crear objetos en concreto. Podemos decir que una clase, es el razonamiento abstracto de un objeto, mientras que el objeto, es su materialización. A la acción de crear objetos, se la denomina "instanciar una clase" y dicha instancia, consiste en asignar la clase, como valor a una variable:

```
class Objeto():
    color = "verde"
    tamanio = "grande"
    aspecto = "feo"
    antenas = Antena()
    ojos = Ojo()
    pelos = Pelo()

    def flotar(self):
        print 12


et = Objeto()
print et.color
print et.tamanio
print et.aspecto
et.color = "rosa"
print et.color
```

Herencia: característica principal de la POO

Como comentamos en el título anterior, algunos objetos comparten las mismas

propiedades y métodos que otro objeto, y además agregan nuevas propiedades y métodos. A esto se lo denomina herencia: una clase que hereda de otra. Vale aclarar, que en Python, **cuando una clase no hereda de ninguna otra, debe hacerse heredar de `object`**, que es la clase principal de Python, que define un objeto.

```
class Antena(object):
    color = ""
    longitud = ""


class Pelo(object):
    color = ""
    textura = ""


class Ojo(object):
    forma = ""
    color = ""
    tamanio = ""


class Objeto(object):
    color = ""
    tamanio = ""
    aspecto = ""
    antenas = Antena()
    ojos = Ojo()
    pelos = Pelo()

    def flotar(self):
        pass


class Dedo(object):
    longitud = ""
    forma = ""
    color = ""


class Pie(object):
    forma = ""
    color = ""
    dedos = Dedo()


# NuevoObjeto sí hereda de otra clase: Objeto
class NuevoObjeto(Objeto):
    pie = Pie()

    def saltar(self):
        pass
```

Accediendo a los métodos y propiedades de un objeto

Una vez creado un objeto, es decir, una vez hecha la instancia de clase, es posible

acceder a su métodos y propiedades. Para ello, Python utiliza una sintaxis muy simple: el nombre del objeto, seguido de punto y la propiedad o método al cuál se desea acceder:

```
objeto = MiClase()
print objeto.propiedad
objeto.otra_propiedad = "Nuevo valor"
variable = objeto.metodo()
print variable
print objeto.otro_metodo()
```

06 Métodos principales del Objeto String

Como comentamos en el capítulo anterior, en Python, todo es un objeto y por tanto, cualquier variable cuyo valor sea de tipo *string*, podrá ser tratada como un subtipo del objeto *string*, el cuál dispone de métodos que son heredados por dicho subtipo.

En este capítulo, veremos los métodos más frecuentes del objeto *string*.

Métodos de formato

Convertir a mayúscula la primera letra

Método: capitalize()

Retorna: una copia de la cadena con la primera letra en mayúsculas

```
>>> cadena = "bienvenido a mi aplicación"
>>> print cadena.capitalize()
Bienvenido a mi aplicación
```

Convertir una cadena a minúsculas

Método: lower()

Retorna: una copia de la cadena en minúsculas

```
>>> cadena = "Hola Mundo"
>>> print cadena.lower()
hola mundo
```

Convertir una cadena a mayúsculas

Método: upper()

Retorna: una copia de la cadena en mayúsculas

```
>>> cadena = "Hola Mundo"
>>> print cadena.upper()
HOLA MUNDO
```

Convertir mayúsculas a minúsculas y viceversa

Método: swapcase()

Retorna: una copia de la cadena convertidas las mayúsculas en minúsculas y viceversa

```
>>> cadena = "Hola Mundo"
>>> print cadena.swapcase()
hOLA mUNDO
```

Convertir una cadena en Formato Título

Método: title()

Retorna: una copia de la cadena convertida

```
>>> cadena = "hola mundo"
>>> print cadena.title()
Hola Mundo
```

Centrar un texto

Método: center(longitud[, “caracter de relleno”])

Retorna: una copia de la cadena centrada

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.center(50, "=")
===========Bienvenido a mi aplicación============
>>> print cadena.center(50, " ")
           Bienvenido a mi aplicación
```

Alinear texto a la izquierda

Método: ljust(longitud[, “caracter de relleno”])

Retorna: una copia de la cadena alineada a la izquierda

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.ljust(50, "=")
Bienvenido a mi aplicación=======================
```

Alinear texto a la derecha

Método: rjust(longitud[, “caracter de relleno”])

Retorna: una copia de la cadena alineada a la derecha

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.rjust(50, "=")
=======================Bienvenido a mi aplicación
>>> print cadena.rjust(50, " ")
                       Bienvenido a mi aplicación
```

Rellenar un texto anteponiendo ceros

Método: zfill(longitud)

Retorna: una copia de la cadena rellena con ceros a la izquierda hasta alcanzar la longitud final indicada

```
>>> numero_factura = 1575
>>> print str(numero_factura).zfill(12)
000000001575
```

Métodos de Búsqueda

Contar cantidad de apariciones de una subcadena

Método: count(“subcadena”[, posicion_inicio, posicion_fin])

Retorna: un entero representando la cantidad de apariciones de *subcadena* dentro de cadena

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.count("a")
3
```

Buscar una subcadena dentro de una cadena

Método: find(“subcadena”[, posicion_inicio, posicion_fin])

Retorna: un entero representando la posición donde inicia la subcadena dentro de cadena. Si no la encuentra, retorna -1

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.find("mi")
13
>>> print cadena.find("mi", 0, 10)
-1
```

Métodos de Validación

Saber si una cadena comienza con una subcadena determinada

Método: startswith("subcadena"[, posicion_inicio, posicion_fin])

Retorna: True o False

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.startswith("Bienvenido")
True
>>> print cadena.startswith("aplicación")
False
>>> print cadena.startswith("aplicación", 16)
True
```

Saber si una cadena finaliza con una subcadena determinada

Método: endswith("subcadena"[, posicion_inicio, posicion_fin])

Retorna: True o False

```
>>> cadena = "bienvenido a mi aplicación".capitalize()
>>> print cadena.endswith("aplicación")
True
>>> print cadena.endswith("Bienvenido")
False
>>> print cadena.endswith("Bienvenido", 0, 10)
True
```

Saber si una cadena es alfanumérica

Método: isalnum()

Retorna: True o False

```
>>> cadena = "pepegrillo 75"
>>> print cadena.isalnum()
False
>>> cadena = "pepegrillo"
>>> print cadena.isalnum()
True
>>> cadena = "pepegrillo75"
>>> print cadena.isalnum()
True
```

Saber si una cadena es alfabética

Método: isalpha()

Retorna: True o False

```
>>> cadena = "pepegrillo 75"
>>> print cadena.isalpha()
False
>>> cadena = "pepegrillo"
>>> print cadena.isalpha()
True
>>> cadena = "pepegrillo75"
>>> print cadena.isalpha()
False
```

Saber si una cadena es numérica

Método: isdigit()

Retorna: True o False

```
>>> cadena = "pepegrillo 75"
>>> print cadena.isdigit()
False
>>> cadena = "7584"
>>> print cadena.isdigit()
True
>>> cadena = "75 84"
>>> print cadena.isdigit()
False
>>> cadena = "75.84"
>>> print cadena.isdigit()
False
```

Saber si una cadena contiene solo minúsculas

Método: islower()

Retorna: True o False

```
>>> cadena = "pepe grillo"
>>> print cadena.islower()
True
>>> cadena = "Pepe Grillo"
>>> print cadena.islower()
False
>>> cadena = "Pepegrillo"
>>> print cadena.islower()
False
>>> cadena = "pepegrillo75"
>>> print cadena.islower()
```

```
True
```

Saber si una cadena contiene solo mayúsculas

Método: isupper()

Retorna: True o False

```
>>> cadena = "PEPE GRILLO"
>>> print cadena.isupper()
True
>>> cadena = "Pepe Grillo"
>>> print cadena.isupper()
False
>>> cadena = "Pepegrillo"
>>> print cadena.isupper()
False
>>> cadena = "PEPEGRILLO"
>>> print cadena.isupper()
True
```

Saber si una cadena contiene solo espacios en blanco

Método: isspace()

Retorna: True o False

```
>>> cadena = "pepe grillo"
>>> print cadena.isspace()
False
>>> cadena = "      "
>>> print cadena.isspace()
True
```

Saber si una cadena tiene Formato De Título

Método: istitle()

Retorna: True o False

```
>>> cadena = "Pepe Grillo"
>>> print cadena.istitle()
True
>>> cadena = "Pepe grillo"
>>> print cadena.istitle()
False
```

Métodos de Sustitución

Dar formato a una cadena, sustituyendo texto dinámicamente

Método: format(*args, **kwargs)

Retorna: la cadena formateada

```
>>> cadena = "bienvenido a mi aplicación {0}"
>>> print cadena.format("en Python")
bienvenido a mi aplicación en Python
>>> cadena = "Importe bruto: ${0} + IVA: ${1} = Importe neto: {2}"
>>> print cadena.format(100, 21, 121)
Importe bruto: $100 + IVA: $21 = Importe neto: 121
>>> cadena = "Importe bruto: ${bruto} + IVA: ${iva} = Importe neto: {neto}"
>>> print cadena.format(bruto=100, iva=21, neto=121)
Importe bruto: $100 + IVA: $21 = Importe neto: 121
>>> print cadena.format(bruto=100, iva=100 * 21 / 100, neto=100 * 21 / 100 + 100)
Importe bruto: $100 + IVA: $21 = Importe neto: 121
```

Reemplazar texto en una cadena

Método: replace("subcadena a buscar", "subcadena por la cual reemplazar")

Retorna: la cadena reemplazada

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print "Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por)
Estimado Sr. Juan Pérez:
```

Eliminar caracteres a la izquierda y derecha de una cadena

Método: strip(["caracter"])

Retorna: la cadena sustituida

```
>>> cadena = "  www.eugeniabahit.com   "
>>> print cadena.strip()
www.eugeniabahit.com
>>> print cadena.strip(' ')
www.eugeniabahit.com
```

Eliminar caracteres a la izquierda de una cadena

Método: lstrip(["caracter"])

Retorna: la cadena sustituida

```
>>> cadena = "www.eugeniabahit.com"
>>> print cadena.lstrip("w." )
eugeniabahit.com
>>> cadena = "       www.eugeniabahit.com"
>>> print cadena.lstrip()
www.eugeniabahit.com
```

Eliminar caracteres a la derecha de una cadena

Método: rstrip([“caracter”])

Retorna: la cadena sustituida

```
>>> cadena = "www.eugeniabahit.com     "
>>> print cadena.rstrip( )
www.eugeniabahit.com
```

Métodos de unión y división

Unir una cadena de forma iterativa

Método: join(iterable)

Retorna: la cadena unida con el iterable (la cadena es separada por cada uno de los elementos del iterable)

```
>>> formato_numero_factura = ("Nº 0000-0", "-0000 (ID: ", ")")
>>> numero = "275"
>>> numero_factura = numero.join(formato_numero_factura)
>>> print numero_factura
Nº 0000-0275-0000 (ID: 275)
```

Partir una cadena en tres partes, utilizando un separador

Método: partition("separador")

Retorna: una tupla de tres elementos donde el primero es el contenido de la cadena previo al separador, el segundo, el separador mismo y el tercero, el contenido de la cadena posterior al separador

```
>>> tupla = "http://www.eugeniabahit.com".partition("www.")
>>> print tupla
('http://', 'www.', 'eugeniabahit.com')
>>> protocolo, separador, dominio = tupla
>>>> print "Protocolo: {0}\nDominio: {1}".format(protocolo, dominio)
Protocolo: http://
Dominio: eugeniabahit.com
```

Partir una cadena en varias partes, utilizando un separador

Método: split("separador")

Retorna: una lista con todos elementos encontrados al dividir la cadena por un separador

```
>>> keywords = "python, guia, curso, tutorial".split(", ")
>>> print keywords
['python', 'guia', 'curso', 'tutorial']
```

Partir una cadena en en líneas

Método: splitlines()

Retorna: una lista donde cada elemento es una fracción de la cadena divida en líneas

```
>>> texto = """Linea 1
Linea 2
Linea 3
Linea 4
"""
>>> print texto.splitlines()
['Linea 1', 'Linea 2', 'Linea 3', 'Linea 4']
>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print texto.splitlines()
['Linea 1', 'Linea 2', 'Linea 3']
```

Ejercicio

Ejercicio Nº1

Crear un módulo para validación de nombres de usuarios. Dicho módulo, deberá cumplir con los siguientes **criterios de aceptación**:

- El nombre de usuario debe contener un mínimo de 6 caracteres y un máximo de 12
- El nombre de usuario debe ser alfanumérico
- Nombre de usuario con menos de 6 caracteres, retorna el mensaje "El nombre de usuario debe contener al menos 6 caracteres"
- Nombre de usuario con más de 12 caracteres, retorna el mensaje "El nombre de usuario no puede contener más de 12 caracteres"
- Nombre de usuario con caracteres distintos a los alfanuméricos, retorna el mensaje "El nombre de usuario puede contener solo letras y números"
- Nombre de usuario válido, retorna True

Ejercicio Nº2

Crear un módulo para validación de contraseñas. Dicho módulo, deberá cumplir con los siguientes **criterios de aceptación**:

- La contraseña debe contener un mínimo de 8 caracteres
- Una contraseña debe contener letras minúsculas, mayúsculas, números y al menos 1 carácter no alfanumérico
- La contraseña no puede contener espacios en blanco
- Contraseña válida, retorna True
- Contraseña no válida, retorna el mensaje "La contraseña elegida no es segura"

Ejercicio Nº3

Crear un módulo que solicite al usuario el ingreso de un nombre de usuario y contraseña y que los valide utilizando los módulos generados en los dos ejercicios anteriores.

Ayuda: para contar la cantidad de caracteres de una cadena, en Python se utiliza la función incorporada: `len(cadena)`

07 Métodos principales del objeto list

En este capítulo, veremos los métodos que posee el objeto *lista*. Algunos de ellos, también se encuentran disponibles para las *tuplas*.

Métodos de agregado

Agregar un elemento al final de la lista

Método: append("nuevo elemento")

```
>>> nombres_masculinos = ["Alvaro", "Jacinto", "Miguel", "Edgardo", "David"]
>>> nombres_masculinos.append("Jose")
>>> print nombres_masculinos
['Alvaro', 'David', 'Edgardo', 'Jacinto', 'Jose', 'Ricky', 'Jose']
```

Agregar varios elementos al final de la lista

Método: extend(otra_lista)

```
>>> nombres_masculinos.extend(["Jose", "Gerardo"])
>>> print nombres_masculinos
['Alvaro', 'David', 'Edgardo', 'Jacinto', 'Jose', 'Ricky', 'Jose', 'Jose',
'Gerardo']
```

Agregar un elemento en una posición determinada

Método: insert(posición, "nuevo elemento")

```
>>> nombres_masculinos.insert(0, "Ricky")
>>> print nombres_masculinos
['Ricky', 'Alvaro', 'David', 'Edgardo', 'Jacinto', 'Jose', 'Ricky', 'Jose',
'Jose', 'Gerardo']
```

Métodos de eliminación

Eliminar el último elemento de la lista

Método: pop()

Retorna: el elemento eliminado

```
>>> nombres_masculinos.pop()
'Gerardo'
>>> print nombres_masculinos
['Ricky', 'Alvaro', 'David', 'Edgardo', 'Jacinto', 'Jose', 'Ricky', 'Jose',
'Jose']
```

Eliminar un elemento por su índice

Método: pop(índice)

Retorna: el elemento eliminado

```
>>> nombres_masculinos.pop(3)
'Edgardo'
>>> print nombres_masculinos
['Ricky', 'Alvaro', 'David', 'Jacinto', 'Jose', 'Ricky', 'Jose', 'Jose']
```

Eliminar un elemento por su valor

Método: remove("valor")

```
>>> nombres_masculinos.remove("Jose")
>>> print nombres_masculinos
['Ricky', 'Alvaro', 'David', 'Jacinto', 'Ricky', 'Jose', 'Jose']
```

Métodos de orden

Ordenar una lista en reversa (invertir orden)

Método: reverse()

```
>>> nombres_masculinos.reverse()
>>> print nombres_masculinos
['Jose', 'Jose', 'Ricky', 'Jacinto', 'David', 'Alvaro', 'Ricky']
```

Ordenar una lista en forma ascendente

Método: sort()

```
>>> nombres_masculinos.sort()
>>> print nombres_masculinos
['Alvaro', 'David', 'Jacinto', 'Jose', 'Jose', 'Ricky', 'Ricky']
```

Ordenar una lista en forma descendente

Método: sort(reverse=True)

```
>>> nombres_masculinos.sort(reverse=True)
>>> print nombres_masculinos
['Ricky', 'Ricky', 'Jose', 'Jose', 'Jacinto', 'David', 'Alvaro']
```

Métodos de búsqueda

Contar cantidad de apariciones elementos

Método: count(elemento)

```
>>> nombres_masculinos = ["Alvaro", "Miguel", "Edgardo", "David", "Miguel"]
>>> nombres_masculinos.count("Miguel")
2
>>> nombres_masculinos = ("Alvaro", "Miguel", "Edgardo", "David", "Miguel")
>>> nombres_masculinos.count("Miguel")
2
```

Obtener número de índice

Método: index(elemento[, indice_inicio, indice_fin])

```
>>> nombres_masculinos.index("Miguel")
1
>>> nombres_masculinos.index("Miguel", 2, 5)
4
```

Anexo sobre listas y tuplas

Conversión de tipos

En el conjunto de las funciones integradas de Python, podemos encontrar dos funciones que nos permiten convertir listas en tuplas y viceversa.

Estas funciones pueden ser muy útiles cuando por ejemplo, una variable declarada como tupla, necesita ser modificada en tiempo de ejecución, para lo cual, debe convertirse en una lista puesto que las tuplas, son inmutables. Lo mismo sucede en el caso contrario: una variable que haya sido declarada como lista y sea necesario convertirla en una colección inmutable.

```
>>> tupla = (1, 2, 3, 4)
>>> tupla
(1, 2, 3, 4)
>>> list(tupla)
[1, 2, 3, 4]
>>> lista = [1, 2, 3, 4]
>>> lista
[1, 2, 3, 4]
>>> tuple(lista)
(1, 2, 3, 4)
```

Concatenación simple de colecciones

A diferencia de otros lenguajes, en Python es muy simple unir varias colecciones de un mismo tipo. Simplemente, se requiere utilizar el operador suma (+) para lograrlo:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista2 = [3, 4, 5, 6, 7, 8]
>>> lista3 = lista1 + lista2
>>> lista3
[1, 2, 3, 4, 3, 4, 5, 6, 7, 8]
>>> tupla1 = (1, 2, 3, 4, 5)
>>> tupla2 = (4, 6, 8, 10)
>>> tupla3 = (3, 5, 7, 9)
>>> tupla4 = tupla1 + tupla2 + tupla3
>>> tupla4
(1, 2, 3, 4, 5, 4, 6, 8, 10, 3, 5, 7, 9)
```

Valor máximo y mínimo

Podemos obtener además, el valor máximo y mínimo tanto de listas como de tuplas:

```
>>> max(tupla4)
10
>>> max(tupla1)
5
>>> min(tupla1)
1
>>> max(lista3)
8
>>> min(lista1)
1
```

Contar elementos

Al igual que para contar caracteres en una string, disponemos de la función integrada `len()` para conocer la cantidad de elementos en una lista o en una tupla:

```
>>> len(lista3)
10
>>> len(lista1)
4
>>> len(tupla2)
4
```

08

Métodos principales del objeto dict

Métodos de eliminación

Vaciar un diccionario

Método: clear()

```
>>> diccionario = {"color": "violeta", "talle": "XS", "precio": 174.25}
>>> print diccionario
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> diccionario.clear()
>>> print diccionario
{}
```

Métodos de agregado y creación

Copiar un diccionario

Método: copy()

```
>>> diccionario = {"color": "violeta", "talle": "XS", "precio": 174.25}
>>> remera = diccionario.copy()
>>> diccionario
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> remera
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> diccionario.clear()
>>> diccionario
{}
>>> remera
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> musculosa = remera
>>> remera
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> musculosa
{'color': 'violeta', 'precio': 174.25, 'talle': 'XS'}
>>> remera.clear()
>>> remera
{}
>>> musculosa
{}
>>>
```

Crear un nuevo diccionario desde las claves de una secuencia

Método: dict.fromkeys(secuencia[, valor por defecto])

```
>>> secuencia = ["color", "talle", "marca"]
>>> diccionario1 = dict.fromkeys(secuencia)
>>> diccionario1
{'color': None, 'marca': None, 'talle': None}
>>> diccionario2 = dict.fromkeys(secuencia, 'valor x defecto')
>>> diccionario2
{'color': 'valor x defecto', 'marca': 'valor x defecto', 'talle': 'valor x
defecto'}
```

Concatenar diccionarios

Método: update(diccionario)

```
>>> diccionario1 = {"color": "verde", "precio": 45}
>>> diccionario2 = {"talle": "M", "marca": "Lacoste"}
>>> diccionario1.update(diccionario2)
>>> diccionario1
{'color': 'verde', 'precio': 45, 'marca': 'Lacoste', 'talle': 'M'}
```

Establecer una clave y valor por defecto

Método: setdefault("clave"[, None|valor_por_defecto])

Si la clave no existe, la crea con el valor por defecto. Siempre retorna el valor para la clave pasada como parámetro.

```
>>> remera = {"color": "rosa", "marca": "Zara"}
>>> clave = remera.setdefault("talle", "U")
>>> clave
'U'
>>> remera
{'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}
>>> remera2 = remera.copy()
>>> remera2
{'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}
>>> clave = remera2.setdefault("estampado")
>>> clave
>>> remera2
{'color': 'rosa', 'estampado': None, 'marca': 'Zara', 'talle': 'U'}
>>> clave = remera2.setdefault("marca", "Lacoste")
>>> clave
'Zara'
>>> remera2
{'color': 'rosa', 'estampado': None, 'marca': 'Zara', 'talle': 'U'}
```

Métodos de retorno

Obtener el valor de una clave

Método: get(clave[, “valor x defecto si la clave no existe”])

```
>>> remera.get("color")
'rosa'
>>> remera.get("stock")
>>> remera.get("stock", "sin stock")
'sin stock'
```

Saber si una clave existe en el diccionario

Método: has_key(clave)

```
>>> existe = remera.has_key("precio")
>>> existe
False
>>> existe = remera.has_key("color")
>>> existe
True
```

Obtener las claves y valores de un diccionario

Método: iteritems() **Alias:** items()

```
diccionario = {'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}

for clave, valor in diccionario.iteritems():
    print "El valor de la clave %s es %s" % (clave, valor)
```

Salida:

```
El valor de la clave color es rosa
El valor de la clave marca es Zara
El valor de la clave talle es U
```

Obtener las claves de un diccionario

Método: keys()

```
>>> diccionario = {'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}
>>> claves = diccionario.keys()
>>> claves
['color', 'marca', 'talle']
```

Obtener los valores de un diccionario

Método: values()

```
>>> diccionario = {'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}
>>> valores = diccionario.values()
>>> valores
['rosa', 'Zara', 'U']
```

Obtener la cantidad de elementos de un diccionario

Para contar los elementos de un diccionario, al igual que con las listas y tuplas, se utiliza la función integrada `len()`

```
>>> diccionario = {'color': 'rosa', 'marca': 'Zara', 'talle': 'U'}
>>> len(diccionario)
3
```

09 El objeto File: trabajando con archivos

Python nos permite trabajar en dos niveles diferentes con respecto al sistema de archivos y directorios. Uno de ellos, es a través del módulo os, que como su nombre lo indica, nos facilita el trabajo con todo el sistema de archivos y directorios, a nivel del propios Sistema Operativo. El segundo nivel -más simple-, es el que nos permite trabajar con archivos, manipulando su lectura y escritura a nivel de la aplicación y tratando a cada archivo como un objeto.

En talleres anteriores, hemos utilizado el objeto file y métodos como `read()`, `readlines()` y `close()`. En este capítulo, nos enfocaremos en este segundo nivel de trabajo, con el fin de conocer al objeto File en mayor profundidad.

Sobre el objeto File

Al igual que sucede con otras variables, manipular una de ellas como un objeto File, es posible, cuando a ésta, se le asigna como valor un archivo.

Para asignar a una variable un valor de tipo file, solo es necesario recurrir a la función integrada **`open()`**, la cuál está destinada a la apertura de un archivo.

La función integrada `open()`, recibe dos **parámetros**:

- El primero de ellos, es **la ruta hacia el archivo** que se desea abrir
- Y el segundo, **el modo** en el cual abrirlo

Modos de Apertura

El **modo de apertura de un archivo**, está relacionado con el objetivo final que responde a la pregunta ***"¿para qué estamos abriendo este archivo?"***. Las respuestas a esta pregunta pueden ser varias. Por ejemplo, podemos querer abrir un archivo para leerlo, para escribirlo, para leerlo y escribirlo, para crearlo si no existe y luego escribir en él, etc.

Es necesario saber, que **cada vez que abrimos un archivo estamos creando un "puntero"**, **el cuál se posicionará dentro del archivo en un lugar determinado** (al comienzo o al final) y **este puntero podrá moverse** dentro de ese archivo, eligiendo su nueva posición, **mediante el número de byte** correspondiente.

Este puntero, se creará -en inicio- dependiendo del modo de apertura indicado, el cuál será indicado a la función `open()` como una *string* en su segundo parámetro. Entre los **modos de apertura posibles**, podemos encontrar los siguientes:

Indicador	Modo de apertura	Ubicación del puntero
`r`	Solo lectura	Al inicio del archivo
`rb`	Solo lectura en modo binario	Al inicio del archivo
`r+`	Lectura y escritura	Al inicio del archivo
`rb+`	Lectura y escritura en modo binario	Al inicio del archivo
`w`	Solo escritura. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`wb`	Solo escritura en modo binario. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo

`w+`	Escritura y lectura. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`wb+`	Escritura y lectura en modo binario. Sobreescribe el archivo si existe. Crea el archivo si no existe.	Al inicio del archivo
`a`	Añadido (agregar contenido). Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
`ab`	Añadido en modo binario (agregar contenido). Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
`a+`	Añadido (agregar contenido) y lectura. Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
`ab+`	Añadido (agregar contenido) y lectura en modo binario. Crea el archivo si éste no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.

Métodos del Objeto File

El objeto file, entre sus métodos más frecuentes, dispone de los siguientes:

Método	Descripción	Uso
`seek(byte)`	Mueve el puntero hacia el byte indicado	archivo = open("remeras.txt", "r") contenido = archivo.read() # el puntero queda # al final del documento **archivo.seek(0)**
`read([bytes])`	Lee todo el contenido de un archivo. Si se le pasa la longitud de bytes, leerá solo el contenido hasta la longitud indicada.	archivo = open("remeras.txt", "r") contenido = **archivo.read()** print contenido
`readline([bytes])`	Lee una línea del archivo.	archivo = open("remeras.txt", "r") linea1 = **archivo.readline()** print linea1
`readlines()`	Lee todas las líneas de un archivo	archivo = open("remeras.txt", "r") for linea in **archivo.readlines()**: print linea
`tell()`	Retorna la posición actual del puntero	archivo = open("remeras.txt", "r") linea1 = archivo.readline() mas = archivo.read(**archivo.tell()** * 2) if **archivo.tell()** > 50: archivo.seek(50)
`write(cadena)`	Escribe *cadena* dentro del archivo	archivo = open("remeras.txt", "r+") contenido = archivo.read() final_de_archivo = archivo.tell() **archivo.write('Nueva linea')** archivo.seek(final_de_archivo) nuevo_contenido = archivo.read() print nuevo_contenido # Nueva linea
`writelines(secuencia)`	Secuencia será cualquier iterable cuyos elementos serán escritos uno por línea	archivo = open("remeras.txt", "r+") contenido = archivo.read() final_de_archivo = archivo.tell() **lista = ['Línea 1\n', 'Línea 2']** **archivo.writelines(lista)** archivo.seek(final_de_archivo) print archivo.readline() # Línea 1 print archivo.readline() # Línea 2
`close()`	Cierra un archivo	archivo = open("remeras.txt", "r") contenido = archivo.read() **archivo.close()** print contenido

Propiedades del objeto file

Se pueden acceder a las siguientes propiedades del objeto file:

- **`closed`**: retorna verdadero si el archivo se ha cerrado. De lo contrario, falso.
- **`mode`**: retorna el modo de apertura.
- **`name`**: retorna el nombre del archivo
- **`encoding`**: retorna la codificación de caracteres de un archivo de texto

```
>>> archivo = open("remeras.txt", "r+")
>>> contenido = archivo.read()
>>> nombre = archivo.name
>>> modo = archivo.mode
>>> encoding = archivo.encoding
>>> archivo.close()
>>> if archivo.closed:
...     print "El archivo se ha cerrado correctamente"
... else:
...     print "El archivo permanece abierto"
...
El archivo se ha cerrado correctamente
>>> nombre
'remeras.txt'
>>> modo
'r+'
>>> encoding
None
```

Cerrando archivos de forma automática

Desde la versión 2.5, Python incorpora una manera “elegante” de trabajar con archivos de forma tal, que se cierren de forma automática sin necesidad de invocar al método `close()`. Se trata de un bloque **`with`**:

```
with open("remeras.txt", "r") as archivo:
    contenido = archivo.read()

print archivo.closed
# True
```

Cuando una estructura `with` finaliza, Python, automáticamente invoca al método `close()`, como se puede ver en el valor de la propiedad `closed`.

Como también se deja ver en el ejemplo, la sentencia `with` utiliza un alias para el objeto file, lo que permite acceder al objeto file, justamente, por el alias indicado.

10 Un Paseo por los Módulos de la librería estándar

Python nos provee de un gran abanico de módulos que integran su librería estándar, como bien puede verse en el manual oficial: http://docs.python.org/modindex.html. En este capítulo, veremos algunos de ellos que se destacan ya sea por la frecuencia de uso como por sus prestaciones.

Módulos de sistema

Entre los módulos de sistema que Python nos provee a través de su librería estándar, podemos destacar tres: **`os`**, **`sys`**, y **`subprocess`.** Haremos una breve reseña de cada uno de ellos, a continuación.

Módulo os

El módulo os nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular la estructura de directorios (para leer y escribir archivos, ver capítulo 9). **Referencia oficial:** http://docs.python.org/library/os.html

Archivos y directorios

El módulo os nos provee de varios métodos para trabajar de forma portable con las funcionalidades del sistema operativo. Veremos a continuación, los métodos más destacados de este módulo.

Descripción	Método
Saber si se puede acceder a un archivo o directorio	`os.access(path, modo_de_acceso)`
Conocer el directorio actual	`os.getcwd()`
Cambiar de directorio de trabajo	`os.chdir(nuevo_path)`
Cambiar al directorio de trabajo raíz	`os.chroot()`
Cambiar los permisos de un archivo o directorio	`os.chmod(path, permisos)`
Cambiar el propietario de un archivo o directorio	`os.chown(path, permisos)`
Crear un directorio	`os.mkdir(path[, modo])`
Crear directorios recursivamente	`os.mkdirs(path[, modo])`
Eliminar un archivo	`os.remove(path)`
Eliminar un directorio	`os.rmdir(path)`
Eliminar directorios recursivamente	`os.removedirs(path)`
Renombrar un archivo	`os.rename(actual, nuevo)`
Crear un enlace simbólico	`os.symlink(path, nombre_destino)`

*Para **ver al módulo `os` trabajando** con funcionalidades del sistema de archivos y directorios, ejecutar **`python os_examples`** de la carpeta `sources` de este capítulo.*

El módulo os y las variables de entorno

El módulo os también nos provee de un diccionario con las variables de entorno relativas al sistema. Se trata del diccionario `environ`:

```
import os

for variable, valor in os.environ.iteritems():
    print "%s: %s" % (variable, valor)
```

os.path

El módulo `os` también nos provee del submódulo `path` (**`os.path`**) el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios. Entre ellas, las más destacadas se describen en la siguiente tabla:

Descripción	Método
Ruta absoluta	`os.path.abspath(path)`
Directorio base	`os.path.basename(path)`
Saber si un directorio existe	`os.path.exists(path)`
Conocer último acceso a un directorio	`os.path.getatime(path)`
Conocer tamaño del directorio	`os.path.getsize(path)`
Saber si una ruta es:	
Una ruta absoluta	`os.path.isabs(path)`
Un archivo	`os.path.isfile(path)`
Un directorio	`os.path.isdir(path)`
Un enlace simbólico	`os.path.islink(path)`
Un punto de montaje	`os.path.ismount(path)`

Para conocer más sobre **`os.path`***, visitar la documentación oficial en* *http://docs.python.org/library/os.path.html*

Módulo sys

El módulo sys es el encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.

Variables del módulo sys

Entre las variables más destacadas podemos encontrar las siguientes:

Variable	Descripción
sys.argv	Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar: `python modulo.py arg1 arg2` `sys.arg` retornará una lista: `['modulo.py', 'arg1', 'arg2']`
sys.executable	Retorna el path absoluto del binario ejecutable del intérprete de Python
sys.maxint	Retorna el número positivo entero mayor, soportado por Python
sys.platform	Retorna la plataforma sobre la cuál se está ejecutando el intérprete
sys.version	Retorna el número de versión de Python con información adicional

Métodos del módulo sys

Entre los métodos más destacados del módulo sys, podemos encontrar los siguientes:

Método	Descripción
sys.exit()	Forzar la salida del intérprete
sys.getdefaultencoding()	Retorna la codificación de caracteres por defecto
Sys.getfilesystemencoding()	Retorna la codificación de caracteres que se utiliza para convertir los nombres de archivos unicode en nombres de archivos del sistema
Sys.getsizeof(object[, default]**)**	Retorna el tamaño del objeto pasado como parámetro. El segundo argumento (opcional) es retornado cuando el objeto no devuelve nada.

Más información sobre el módulo **sys***, puede obtenerse en* *http://docs.python.org/library/sys.html*

Módulo subprocess

El módulo `subprocess` es aquel que nos permite trabajar de forma directa con órdenes del sistema operativo.

> *El módulo* **`subprocess`** *se presenta en este capítulo solo con fines educativos, mostrando ejemplos básicos y sencillos. Por lo tanto,* ***se recomienda tener mucho cuidado en el uso de este módulo, desaconsejando su uso para órdenes que puedan comprometer el sistema.***

Entre los métodos más comunes de `subprocess`, podemos encontrar `subprocess`**`.call().`** Este método, suele ser útil, para ejecutar órdenes sencillas, como por ejemplo, **limpiar la pantalla**:

```
from subprocess import call

call('clear')
```

El método `call`, esperará recibir como primer argumento, el comando a ser ejecutado, como se mostró en el ejemplo anterior. Sin embargo, si el comando requiere argumentos, como primer parámetro, `call` necesitará recibir una lista donde el primer elemento será el comando y el segundo, sus argumentos. Un ejemplo de ello, es el siguiente código encargado de hacer un **listado de archivos y directorios**:

```
from subprocess import call

comando_y_argumentos = ['ls', '-lha']
call(comando_y_argumentos)
```

El módulo `subprocess` también nos provee del submódulo **`Popen`**, el cuál nos permite, no solo ejecutar órdenes al igual que `call`, sino mantener un mejor control sobre las salidas.

Capturando la salida con Popen

El manejo y captura de las salidas, puede resultar un poco complejo. Por eso, intentaremos explicarlo paso a paso a fin de evitar confusiones.

Lo primero que debemos tener en cuenta, es que Popen (al igual que call), como primer argumento, recibirá el comando a ser ejecutado o una lista de dos elementos, donde el primero sea el comando y el segundo, la lista de argumentos. Hasta aquí, no hay diferencia con call. Sin embargo, si la salida del proceso abierto con Popen no es tratada, el proceso quedará abierto.

Por ejemplo, **el siguiente código quedaría en ejecución**:

```
from subprocess import Popen
Popen(['ls', '-lha']
```

A diferencia de call, Popen no es un método de subprocess, sino, un objeto. Cómo tal, la forma correcta de iniciar un proceso con Popen, será entonces, crear un objeto Popen para poder acceder a sus métodos, y así evitar, que el proceso quede abierto en ejecución. De esta forma, creamos el objeto y luego, llamamos al método **wait()** de Popen, el cual **espera a que el proceso finalice**.

```
from subprocess import Popen

proceso = Popen(['ls', '-lha'])
proceso.wait()
```

Si ejecutamos el código anterior, al igual que sucede con call, la salida obtenida es directamente plasmada en pantalla. Es aquí donde entra en juego, el manejo de las salidas que Popen nos permite hacer.

Entradas y salidas que pueden ser capturadas con Popen

stdout

nomenclatura correspondiente a la salida estándar en sistemas UNIX-Like. Es la encargada de almacenar la salida de un programa.

stdin

nomenclatura correspondiente a la entrada estándar en sistemas UNIX-like. Es la encargada de enviar información a un programa.

stderr

al igual que las anteriores, se utiliza como referencia a los errores producidos en la salida de un programa.

Utilizando tuberías para capturar la salida

Popen nos permite capturar tanto la entrada como la salida estándar o su error. Para

efectuar dicha captura, tanto **`stdout`** como **`stdin`** y/o **`stderr` se pasan como argumentos clave a `Popen`. El valor de dichos argumentos, deberá ser** un archivo o **una tubería** que funcione como tal. Y para esto, `Popen`, también nos provee de una tubería para capturar dichas entradas y salidas, llamada **`PIPE`**.

De esta forma, si quisiéramos capturar la salida estándar o error de nuestro código, debemos pasarle a `Popen`, `stdout` y `stderr` como argumentos claves, con `PIPE` como valor de cada uno de ellos, para lo cual, también debemos importar `PIPE`:

```
from subprocess import PIPE, Popen

proceso = Popen(['ls', '-lha'], stdout=PIPE, stderr=PIPE)
```

Al capturar la salida estándar en una tubería, ya no será necesario poner en espera al proceso, puesto que directamente será capturado por la tubería, permitiéndonos acceder a la lectura tanto de `stdout` como de `stderr`, como si se tratara de cualquier otro archivo:

```
proceso = Popen(['ls', '-lha'], stdout=PIPE, stderr=PIPE)
error_econtrado = proceso.stderr.read()
listado = proceso.stdout.read()
```

Capturando la salida, como bien se puede ver en el ejemplo, `stdout` y `stderr`, son tratados como archivos (de hecho, lo son ya que hemos utilizado una tubería). Por lo tanto, deben ser cerrados una vez leídos:

```
proceso = Popen(['ls', '-lha'], stdout=PIPE, stderr=PIPE)
error_econtrado = proceso.stderr.read()
proceso.stderr.close()
listado = proceso.stdout.read()
proceso.stdout.close()
```

Luego, podremos manipular dichas lecturas, como cualquier *string*:

```
if not error_encontrado:
    print listado
else:
    print "Se produjo el siguiente error:\n%s" % error_encontrado
```

> *Para conocer más sobre* **`subprocess`**, *ingresar en* *http://docs.python.org/library/subprocess.html*

Módulos para el programador

Debuguear código con Pdb

El módulo pdb nos sirve para controlar paso a paso, la ejecución de nuestros programas. Pdb se utiliza solo para debuguear y su utilidad consiste en permitirnos conocer el lugar exacto y el por qué, nuestro script falla.

Imagina que tienes un archivo que genera errores y no logras descubrir la solución. Puedes importar el módulo pdb y hacer una llamada a **`pdb.Pdb().set_trace()`** en la línea, a partir de la cuál, deseas que tu script comience a “caminar paso a paso” para así, descubrir el error:

```
# -*- coding: utf-8 -*-
import pdb
from subprocess import call, Popen, PIPE

# Limpiar la pantalla
call("clear")

pdb.Pdb().set_trace()
proceso = Popen(['ls', '-lha'], stdout=PIPE, stderr=PIPE)
error_encontrado = proceso.stderr.read()
proceso.stderr.close()
listado = proceso.stdout.read()
proceso.stdout.close()

if not error_encontrado:
    print listado
else:
    print "Se produjo el siguiente error:\n%s" % error_encontrado
```

A partir de la línea donde `pdb.Pdb().set_trace()` se encuentra, al ejecutar `python tu_archivo.py`, Pdb comenzará a ejecutar tu archivo línea por línea, esperando tu respuesta. Por ejemplo, en el código anterior, tras la ejecución del archivo, la pantalla se limpiará y Pdb comenzará a actuar, mostrándote la línea que sigue:

```
-> proceso = Popen(['ls', '-lha'], stdout=PIPE, stderr=PIPE)
(Pdb)
```

Pdb estará esperando tu orden para continuar. Entre las órdenes más usuales, puedes indicar:

```
n    (next) ejecuta el código mostrado y salta a la siguiente línea de tu archivo
s    (step) te mostrará paso a paso el camino recorrido
            hasta poder ejecutar la siguiente línea de tu archivo
c    (continue) ejecuta el archivo hasta encontrar un punto de quiebre
q    (quit) abandonar el debugger
```

Pero no solo un comando, puede ser ordenado a `Pdb.` También es posible, "depurar" el código de tu archivo, ejecutando alguna instrucción:

```
-> listado = proceso.stdout.read()
(Pdb) n
> /home/eugenia/Cursos/Python para Principiantes/Módulo
5/sources/subprocess_examples/examples.py(13)<module>()
-> proceso.stdout.close()
(Pdb) listado.upper()
'TOTAL 12K\nDRWXRWXR-X 2 EUGENIA EUGENIA 4,0K 2012-07-07 17:34 .\nDRWXRWXR-X 8
EUGENIA EUGENIA 4,0K 2012-07-07 17:34 ..\n-RW-RW-R-- 1 EUGENIA EUGENIA  429
2012-07-07 20:48 EXAMPLES.PY\n'
(Pdb)
```

Puedes obtener más información sobre **Pdb**
ingresando en [*http://docs.python.org/library/pdb.html*](http://docs.python.org/library/pdb.html)

Documentar tu app con pydoc

Con tan solo colocar los *docstrings* correspondientes en cada módulo y/o función de tu Python app, ejecutar en la termina **`pydoc tu_app`** será suficiente para acceder a toda la documentación:

```
~$ pydoc tienda
Help on package tienda:

NAME
    tienda

FILE
    /home/eugenia/pythonapps/tienda/__init__.py

PACKAGE CONTENTS
    __main__
    administracion (package)
    buscador (package)
    core (package)
```

Alternativamente, también puedes obtener la documentación en formato HTML. Para ello, deberás ejecutar: **`pydoc -w nombre_del_modulo`**

Otra alternativa, es iniciar un servidor Web local, que te permita navegar por la documentación de tu app. Para ello, simplemente ejecuta **`pydoc -p`** ***`n`*** (donde n, es el número del puerto por el cual accederás. Por ejemplo, **`pydoc -p 8080`** inicia el servidor en `http://localhost:8080/`).

Probar el código antes de enviarlo a producción con doctest

El módulo `doctest` de Python, nos permite indicar fragmentos de código en los comentarios de nuestros módulos, que emulen instrucciones del intérprete interactivo, ejecutándolas de forma tal, que podamos automatizar las pruebas de nuestra aplicación.

```
import doctest


def sumar_dos_numeros(a, b):
    """Suma dos números y retorna su resultado

    Argumentos:
    a -- primer sumando
    b -- segundo sumando

    Test:
    >>> sumar_dos_numeros(25, 10)
    35
    >>> sumar_dos_numeros(30, 20)
    50
    """
    return a + b

if __name__ == "__main__":
    doctest.testmod()
```

Si vemos el texto debajo de "Test:", luce como el intérprete interactivo.

Aquí estoy invocando a la función:

```
>>> sumar_dos_numeros(25, 10)
```

Y debajo, estoy "simulando" el resultado que arrojaría en el intérprete interactivo. Esto, será interpretado por `doctest`, como "el resultado esperado":

```
35
```

Para correr los test, solo bastará con ejecutar:

```
~$ python modulo.py -v
```

Y obtendremos un resultado similar a:

```
eugenia@cochito:~/pythonapps/doctest_examples$ python suma.py -v
Trying:
    sumar_dos_numeros(25, 10)
Expecting:
    35
ok
```

```
Trying:
    sumar_dos_numeros(30, 20)
Expecting:
    50
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.sumar_dos_numeros
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Para saber más sobre **`doctest`**, *visita*
http://docs.python.org/library/doctest.html

Lectura complementaria: http://www.maestrosdelweb.com/editorial/guia-python-testeando-codigo-doctest-comentarios/

Módulos que resuelven necesidades funcionales

Obtener datos aleatorios

Con el módulo **random** de la librería estándar de Python, es posible obtener datos aleatorios. Entre los métodos de los cuáles dispone, se destacan los siguientes:

Método	Descripción
random.randint(a, b**)**	Retorna un número aleatorio entero entre a y b
random.choice(secuencia**)**	Retorna cualquier dato aleatorio de secuencia
random.shuffle(secuencia**)**	Retorna una mezcla de los elementos de una secuencia
random.sample(secuencia, n**)**	Retorna n elementos aleatorios de secuencia

```
import random

# Generar números aleatorios entre 49999 y 99999
lista = []

for n in range(0, 50):
    lista.append(random.randint(49999, 99999))


# Elegir un número al azar
numero_al_azar = random.choice(lista)

# Elegir 5 números al azar
numeros_al_azar = random.sample(lista, 5)

# reordenar los elementos de una lista
mujeres = ["Ana", "Beatriz", "Camila", "Carmen", "Delia", "Dora", "Emilse"]
random.shuffle(mujeres)
```

Para conocer más sobre **random**, *ingresar en* *http://docs.python.org/library/random.html*

Wrapear un texto

El módulo **`textwrap`**, entre muchas otras funcionalidades, a través del método `wrap()`, nos permite *wrapear* un texto extenso, obteniendo una lista con cada línea de texto conservando la longitud deseada:

```
textwarp.wrap(texto, 80)
```

Retorna una lista donde cada elemento será una línea de texto, de longitud no superior a los 80 caracteres.

```
import textwrap

texto = "Lorem ipsum ad his scripta blandit partiendo, eum fastidii accumsan
euripidis in, eum liber hendrerit an. Qui ut wisi vocibus suscipiantur, quo
dicit ridens inciderint id. Quo mundi lobortis reformidans eu, legimus
senserit definiebas an eos. Eu sit tincidunt incorrupte definitionem, vis
mutat affert percipit cu, eirmod consectetuer signiferumque eu per. In usu
latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus
placerat per.”

wraps = textwrap.wrap(texto, 60)
for linea in wraps:
    print linea
```

Para conocer más sobre **`textwrap`** *ingresar en*
http://docs.python.org/library/textwrap.html

Módulos e Internet

Acceder al navegador Web

Abrir una URL en una nueva pestaña del navegador:

```
import webbrowser
webbrowser.open_new_tab("http://www.eugeniabahit.com")
```

Más sobre **`webbrowser`** *en*
http://docs.python.org/library/webbrowser.html

`urllib2` *es otro módulo interesante para manipular peticiones HTTP e interactuar a través de Internet:*
http://docs.python.org/library/urllib2.html

Conectarse vía FTP

El módulo **`ftplib`** de la librería estándar de Python, nos provee de los métodos necesarios para crear clientes FTP de forma rápida y sencilla.

Conectarse a un servidor FTP

Para conectarse a un servidor `FTP`, el módulo `ftplib` nos provee de la clase `FTP`. El método constructor de la clase `FTP` (método `__init__()`), recibe como parámetros al host, usuario, clave, de forma tal que pasando estos parámetros durante la instancia a FTP, se ahorra el uso de los métodos `connect(host, port, timeout)` y `login(user, pass)`.

```
from ftplib import FTP

# Conectarse con los métodos connect y login
ftp = FTP()
ftp.connect('66.228.52.93', 21, -999)
ftp.login('miuser', 'miclave')

# Conectarse en la instancia a FTP
ftp = FTP('66.228.52.93', 'miuser', 'miclave')
```

La clase FTP, se compone -entre otros- de los siguientes métodos:

Método	Descripción
FTP.connect(host[, puerto, timeout]**)**	Se conecta al servidor FTP
FTP.login(user, pass**)**	Se loguea en el servidor
FTP.close()	Finaliza la conexión
FTP.set_pasv(bool**)**	Establece la conexión en modo pasivo si el parámetro es True.
FTP.getwelcome()	Retorna el mensaje de bienvenida del servidor
FTP.dir()	Retorna un listado de archivos y directorios de la carpeta actual
FTP.cwd(path**)**	Cambia el directorio de trabajo actual a path
FTP.mkd(path**)**	Crea un nuevo directorio
FTP.pwd()	Retorna el directorio de trabajo actual
FTP.rmd(path**)**	Elimina el directorio path
FTP.storlines('STOR destino**', open(**localfile, **'r')**	Lee localfile y lo escribe en destino
FTP.rename(actual, nuevo**)**	Renombra el archivo "actual" por "nuevo"
FTP.delete(filename**)**	Elimina un archivo
FTP.retrlines('RETR archivo_remoto**')**	Lee archivo_remoto y retorna su contenido

```
# -*- coding: utf-8 -*-
from ftplib import FTP

ftp = FTP()
ftp.connect('66.228.52.93', 21, -999)
ftp.login('user', 'pass')
print ftp.getwelcome()
ftp.mkd('nuevo-dir')
ftp.cwd('nuevo-dir')
print ftp.pwd()
ftp.storlines('STOR example.txt', open('ftp_examples.py', 'r'))
ftp.rename('example.txt', 'example.py')
ftp.dir()
archivo = ftp.retrlines('RETR example.py')
print archivo
ftp.close()
```

Para conocer más sobre **`ftplib`**, *ingresar en* *http://docs.python.org/library/ftplib.html*

11 Introducción a MySQL y el lenguaje SQL

En este capítulo haremos una introducción a conceptos elementales sobre bases de datos, MySQL y el lenguaje de consulta SQL.

Acerca de MySQL

MySQL es un **servidor de Bases de Datos SQL** (Structured Query Language) que se distribuye en dos versiones:

- Una versión GPL (Software Libre)
- Otra versión privativa, llamada MySQL AB

En este curso, utilizaremos la versión estándar licenciada bajo la GNU General Public License (GPL). Puedes descargar el **manual completo de MySQL** en el siguiente enlace: http://downloads.mysql.com/docs/refman-5.0-es.a4.pdf

Instalación y configuración de MySQL

Para instalar MySQL, por línea de comandos, escribe:

```
sudo apt-get install mysql-server mysql-client
```

Durante la instalación, el sistema te pedirá que ingreses una contraseña para la administración de MySQL. Asigna una contraseña que puedas recordar fácilmente y mantenla a salvo ya que deberás utilizarla frecuentemente.

Una vez que finalice la instalación, ejecuta el siguiente comando a fin de securizar el servidor MySQL (esta configuración, es válida también, para servidores de producción):

```
sudo mysql_secure_installation
```

A continuación, el sistema te pedirá que ingreses la contraseña actual para administración de MySQL (la del usuario root de MySQL). Ten en cuenta que la contraseña no será mostrada mientras escribes:

```
Enter current password for root (enter for none):
```

A continuación, te preguntará si deseas modificar esa contraseña. Salvo que desees modificarla, ingresa n:

```
Change the root password? [Y/n] n
```

Ahora la pregunta, será si deseas eliminar usuarios anónimos. Responde que sí:

```
Remove anonymous users? [Y/n] Y
```

Luego, te preguntará si desees desabilitar el acceso remoto al usuario root de MySQL. Por supuesto, responde que sí:

```
Disallow root login remotely? [Y/n] Y
```

La siguiente pregunta será si deseas eliminar la base de datos de prueba y el acceso a ella. También responde que sí:

```
Remove test database and access to it? [Y/n] Y
```

Finalmente, te preguntará si deseas recargar las tablas de privilegios (esto es para asegurar que todos los cambios realizados surjan efecto). Entonces, responde sí, por última vez:

```
Reload privilege tables now? [Y/n] Y
```

Iniciar, reiniciar y detener el servidor MySQL

En ocasiones necesitarás iniciar, reiniciar o detener el servidor de bases de datos, MySQL.

Las **opciones** disponibles son:

```
     stop    detiene el servidor
    start    inicia el servidor
  restart    reinicia el servidor
```

Para iniciar, reiniciar o detener el servidor, deberás **ejecutar el siguiente comando**, seguido de la opción deseada:

```
sudo /etc/init.d/mysql opcion_deseada
```

Lógicamente reemplazando `opcion` por `stop`, `start` o `restart` según si deseas parar, iniciar o reiniciar el servidor.

Administración de MySQL

Una vez que comencemos a utilizar bases de datos, necesitarás poder acceder a las opciones de administración de las mismas. Por lo tanto, te recomiendo tener siempre a mano este capítulo, para poder consultarlo con frecuencia.

Conectarse y desconectarse al servidor

Para conectarte deberás ejecutar el siguiente comando:

```
mysql -u root -p
```

A continuación, deberás ingresar la contraseña del root de MySQL (no es la del root del SO. Es la que hemos configurado durante la instalación de MySQL).

Las `-u` y `-p` significan usuario y password respectivamente.

Te aparecerá un shell interactivo para MySQL:

```
mysql>
```

Allí podremos escribir los comandos necesarios para administrar el servidor de bases de datos.

Comandos para administrar MySQL dede el shell interactivo

La siguiente tabla describe los comandos de uso frecuente que necesitarás para administrar el servidor de bases de datos desde el shell interactivo.

Es una buena idea, imprimir esta tabla para tenerla siempre a mano :)

COMANDO	DESCRIPCIÓN
`show databases;`	Muestra todas las bases de datos creadas en el servidor
`use nombre_de_la_base_de_datos;`	Indicar que vas a comenzar a utilizar la base de datos elegida
`create database nombre_de_la_db;`	Crear una nueva base de datos
`quit`	Salir del shell interactivo

Sobre el lenguaje SQL

SQL -siglas de *Structured Query Language*-, es el **lenguaje de consultas a bases de datos**, que nos permitirá crear, modificar, consultar y eliminar tanto bases de datos como sus tablas y registros, desde el shell interactivo de MySQL y también desde Python.

Como todo **lenguaje informático**, posee su propia **sintaxis**, **tipos de datos** y **elementos**.

En este curso, **abordaremos los conceptos básicos sobre SQL** que nos permitan desarrollar aplicaciones de media complejidad, sin profundizar en el lenguaje en sí, sino solo en aquellos **aspectos mínimamente necesarios relacionados con MySQL**.

Tipos de datos más comunes (recomendados)

La siguiente tabla, muestra los tipos de datos más comunes, aceptados por versiones la versión 5.0.3 o superior, de MySQL.

Tipo de dato	Denominación	Especificaciones	Ejemplo
Entero	**INT(N)**	N = cantidad de dígitos	INT(5)
Número decimal	**DECIMAL(N, D)**	N = cantidad de dígitos totales D = cantidad de decimales	DECIMAL(10, 2)
Booleano	**BOOL**		BOOL
Fecha	**DATE**		DATE
Fecha y hora	**DATETIME**		DATETIME
Fecha y hora automática	**TIMESTAMP**		TIMESTAMP
Hora	**TIME**		TIME
Año	**YEAR(D)**	D = cantidad de dígitos (2 o 4)	YEAR(4)
Cadena de longitud fija	**CHAR(N)**	N = longitud de la cadena – entre 0 y 255	CHAR(2)
Cadena de longitud variable	**VARCHAR(N)**	N = longitud máxima de la cadena – entre 0 y 65532	VARCHAR(100)
Bloque de texto de gran longitud variable	**BLOB**		BLOB

Sintáxis básica de las sentencias SQL

Una sentencia SQL (denominada *"query"* en la jerga informática), es una **instrucción** escrita en lenguaje SQL. Veremos aquí, el tipo de sentencias más habituales.

Crear tablas en una base de datos

Sintaxis:

```
CREATE TABLE nombre_de_la_tabla(
    nombre_del_campo TIPO_DE_DATO,
    nombre_de_otro_campo TIPO_DE_DATO
);
```

Ejemplo:

```
CREATE TABLE productos(
    producto VARCHAR(125),
    descripcion BLOB,
    precio DECIMAL(6, 2),
    en_stock BOOL
);
```

Explicación:

```
CREATE TABLE productos
```

Crear una nueva tabla llamada “productos”

```
producto VARCHAR(125),
```

Crear un campo llamado producto, de tipo cadena de texto de longitud variable, con una longitud máxima de 125 caracteres

```
descripcion BLOB,
```

Crear un campo llamado descripción, de tipo bloque de texto de gran longitud

```
precio DECIMAL(6, 2),
```

Crear un campo precio de tipo numérico de longitud máxima de 6 digítos de los cuales, solo 2 pueden ser decimales

```
en_stock BOOL
```

Crear un campo llamado “en_stock” del tipo booleano

Insertar datos en una tabla

Sintaxis:

```
INSERT INTO
    nombre_de_la_tabla(campo1, campo2, campo10..)
    VALUES(dato1, dato2, dato10...);
```

Ejemplo:

```
INSERT INTO
    productos(producto, precio, en_stock)
    VALUES('Bolsa de dormir para alta montaña', 234.65, TRUE);
```

Explicación:

```
INSERT INTO
    productos(producto, precio, en_stock)
```

Insertar un nuevo registro en los campos producto, precio y en_stock de la tabla productos

```
VALUES('Bolsa de dormir para alta montaña', 234.65, TRUE);
```

Con los valores "Bolsa de dormir para alta montaña", 234.65 y verdadero, respectivamente en cada uno de los campos indicados

Seleccionar registros

Sintaxis:

```
SELECT    campo1, campo2, campo10
FROM      tabla;
```

Ejemplo:

```
SELECT    producto, precio
FROM      productos;
```

Explicación:

```
SELECT    producto, precio
```

Seleccionar los campos producto y precio

```
FROM      productos;
```

De la tabla productos

Modificar registros

Sintaxis:

```
UPDATE  tabla
SET     campo1 = valor,
        campo2 = valor,
        campo10 = valor;
```

Ejemplo:

```
UPDATE     productos
SET        en_stock = FALSE,
           precio = 0;
```

Explicación:

```
UPDATE     productos
```

Actualizar la tabla productos

```
SET        en_stock = FALSE,
```

Modificar el campo en_stock por falso

```
precio = 0;
```

y el campo precio a 0

Eliminar registros

Sintaxis:

```
DELETE FROM tabla;
```

Ejemplo:

```
DELETE FROM productos;
```

Explicación:

```
DELETE FROM productos;
```

Eliminar todos los registros de la tabla productos

Consultas avanzadas

Si bien no veremos aquí consultas realmente complejas, ya que el curso se basa en el lenguaje de programación Python y no, en el lenguaje de consulta SQL, haremos un rápido paseo, por las opciones disponibles en SQL para sentencias más complejas que las anteriores.

La cláusula WHERE

Las sentencias en SQL, se componen de **cláusulas**. Y `WHERE` es una de ellas. La **cláusula `WHERE`** nos permite **filtrar registros en una sentencia** SQL.

Esta cláusula, funciona de forma similar a la comparación de expresiones en Python, utilizando los siguientes **operadores de comparación**:

```
>        mayor que                 <        menor que
=        igual que                 <>       distinto que
>=       mayor o igual que          <=       menor o igual que

BETWEEN n1 AND n2            entre n1 y n2
IS NULL|TRUE|FALSE           es nulo | es verdadero | es falso
IN(valor1, valor2, va...)    contiene
```

Por supuesto, tambien admite **operadores lógicos**:

```
AND (y)          NOT (negación)          OR (o)
```

Veamos algunos ejemplos:

Seleccionar productos donde precio sea menor que 1000:

```
SELECT    producto,
          precio

FROM      productos
WHERE     precio < 1000;
```

Aumentar el 10% del precio de los productos, que actualmente se encuentren entre 150 y 200:

```
UPDATE    productos
SET       precio = (precio * 1.10)
WHERE     precio BETWEEN 150 AND 200;
```

Seleccionar productos donde en_stock no sea falso

```
SELECT    producto,
          precio
FROM      productos
WHERE     en_stock IS NOT FALSE;
```

Eliminar productos cuyos precios sean 100, 200 y/o 300 y además, en_stock sea falso o producto sea nulo:

```
DELETE
FROM      productos
WHERE     precio IN(100, 200, 300)
AND       (en_stock IS FALSE
OR        producto IS NULL);
```

Modificar en_stock a verdadero donde precio sea menor que 50 y producto no sea nulo:

```
UPDATE    productos
SET       en_stock = TRUE
WHERE     precio < 50
AND       en_stock IS NOT NULL;
```

Ordenando consultas: la cláusula ORDER BY

Es posible además, ordenar los resultados de una consulta, en forma **ascendente** (ASC) o **descendente** (DESC):

```
SELECT    producto,
          descripcion,
          precio

FROM      productos

WHERE     precio BETWEEN 1 AND 50
```

```
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC;
```

También es posible, ordenar los resultados de la consulta, por más de un campo:

```
SELECT    producto,
          descripcion,
          precio

FROM      productos

WHERE     precio BETWEEN 1 AND 50
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC,
          producto ASC;
```

Alias de tablas y campos

Otra posibilidad que nos da el lenguaje SQL, es utilizar alias para el nombre de los campos y las tablas. Estos alias se asignan mediante la palabra clave reservada, **AS**:

```
SELECT    producto       AS 'Nombre del Producto',
          descripcion    AS Detalles,
          precio         AS Importe

FROM      productos      AS p

WHERE     precio BETWEEN 1 AND 50
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC,
          producto ASC;
```

> *Nótese que los **alias que contengan caracteres extraños**, deben ser **encerrados entre comillas simples***

Funciones del lenguaje SQL de MySQL

Es posible también, utilizar diversas funciones propias del lenguaje SQL -ya sea estandar o de MySQL- a fin de poder obtener los datos con cierto formato. Veremos aquellas de uso más frecuente.

Contar la cantidad de registros: COUNT()

```
SELECT    COUNT(producto)   AS Cantidad
FROM      productos;
```

Sumar totales: SUM()

```
SELECT    SUM(precio)   AS Total
FROM      productos;
```

Concatenar cadenas: CONCAT()

```
SELECT    producto,
          CONCAT('USD ', precio, '.-')    AS Precio
FROM      productos;
```

Nótese que las cadenas de caracteres deben encerrarse entre comillas simples y que el operador de concatenación para esta función, es la coma.

Convertir a minúsculas y mayúsculas: LCASE() y UCASE()

```
SELECT    UCASE(producto),
          LCASE(descripcion)
FROM      productos;
```

Reemplazar datos: REPLACE()

```
SELECT    REPLACE(descripcion, '\n', '<br/>')    AS Descripcion
FROM      productos;
```

Reemplaza '\n' por '
'

Obtener los primeros o últimos caracteres: LEFT() y RIGHT()

```
SELECT    LEFT(producto, 50)
FROM      productos;
```

Redondear números: ROUND()

```
SELECT    ROUND(precio, 2)
FROM      productos;
```

Retornará los precios con 2 decimales

Obtener solo la fecha de un campo DATETIME o TIMESTAMP: DATE()

```
SELECT    DATE(campo_datetime)
FROM      tabla;
```

Obtener una fecha formateada: DATE_FORMAT()

```
SELECT    DATE_FORMAT(campo_fecha, '%d/%m/%Y')
FROM      tabla;
```

Obtener el registro con el valor máximo y mínimo: MAX() y MIN()

```
SELECT    MAX(precio)
FROM      productos;
```

Retorna el producto con el precio más caro

```
SELECT    MIN(precio)
FROM      productos;
```

Retorna el producto con el precio más barato

Optimización de bases de Datos

A continuación, encontrarás una lista de consejos que SIEMPRE debes seguir, al momento de crear nuevas tablas y escribir sentencias SQL.

Todos los registros deben tener un ID único

Cuando crees tablas, asígnales un campo `id` de tipo autonumérico incremental y establécelo como índice primario. Cuando agregues registros, este campo se completará automáticamente, con un número incremental, que te servirá para optimizar tus consultas y contar con un campo que te permita reconocer el registro como único.

```
CREATE TABLE productos(
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    producto VARCHAR(125)
);
```

El campo `id`, será como cualquier otro y lo podrás seleccionar en un SELECT o utilizarlo e cualquier cláusula WHERE.

Crear índices en las tablas

Todas las tablas deben tener un índice. El índice se asigna a uno o más campos, y es utilizado por SQL para filtrar registros de forma más rápida. Debes crear índices con precaución, ya que de la misma forma que se aceleran las consultas, se retrasa la inserción y actualización de registros, puesto que la base de datos, deberá actualizar los índices cada vez que se agreguen o modifiquen datos.

Cuando una consulta es ejecutada, MySQL tratará de encontrar primero la respuesta en los campos índice, y lo hará en el orden que los índices hayan sido creados.

¿Cuándo agregar índices? Cuando vayas a utilizar una combinación de campos en la cláusula WHERE. Por ejemplo, si filtrarás a menudo, los datos de la tabla producto por su campo precio y en_stock, que precio y en_stock sean un índice de múltiples campos:

```
CREATE TABLE productos(
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    producto VARCHAR(125),
    precio DECIMAL(10, 2),
    en_stock BOOL,
    descripcion BLOB,
    INDEX(precio, en_stock)
);
```

Indica cuáles campos no pueden ser nulos

SQL te da la posibilidad de indicar qué campos no pueden estar nulos. Indicar que un campo no debe estar nulo, te ayudará a no almacenar registros defectuosos en tu base de datos.

```
CREATE TABLE productos(
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    producto VARCHAR(125) NOT NULL,
    precio DECIMAL(10, 2) NOT NULL,
    en_stock BOOL,
    descripcion BLOB NOT NULL,
    INDEX(precio, en_stock)
);
```

Utiliza el motor InnoDB

El motor de bases de datos InnoDB, te permitirá crear tablas relaciones optimizando su rendimiento. Al momento de crear tus tablas, indica que utilizarás el motor InnoDB:

```
CREATE TABLE productos(
```

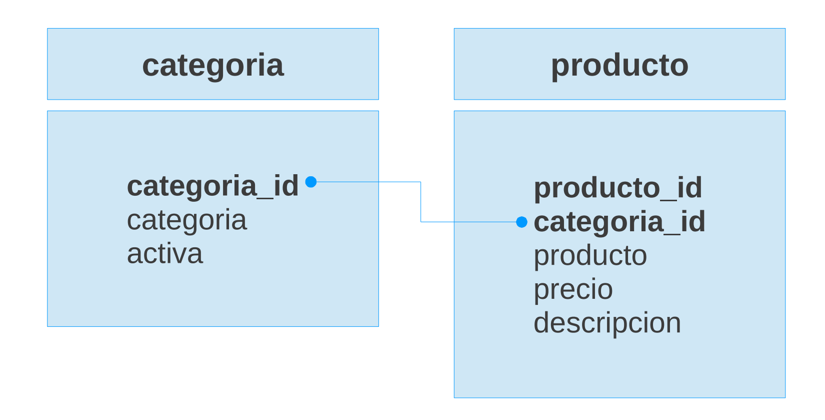
```
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    producto VARCHAR(125) NOT NULL,
    precio DECIMAL(10, 2) NOT NULL,
    en_stock BOOL,
    descripcion BLOB NOT NULL,
    INDEX(precio, en_stock)
) ENGINE=InnoDB;
```

Aprende más sobre el lenguaje SQL, gratis en 1KeyData

Bases de datos relacionales

Así como en la orientación a objetos, algunas clases se relacionan con otras, ya sea a través de la herencia o la composición, cuando nuestros objetos deben guardar un almacén de datos, esa relación, debe conservarse también en la base de datos de nuestra aplicación.

Si te fijas el siguiente esquema, puede entenderse como dos objetos con sus propiedades y a la vez, como dos tablas, relacionadas entre sí:



El objeto producto, se relaciona directamente con el objeto categoría. Esto significa que nuestros productos, pertenecen a una categoría determinada. Se relacionan a través del campo-propiedad, categoria_id.

Para crear bases de datos relacionales, **primero debemos crear nuestros modelos**, a fin de obtener las relaciones que serán necesarias:

```
class Categoria(object):

    categoria_id = 0;
    categoria = ""
    activa = True


class Producto(object):

    producto_id = 0
    categoria = Categoria()
    producto = ""
    precio = 0.0
    descripcion = ""
```

Una vez que tenemos los modelos, podemos pasar a **crear las tablas**:

```
CREATE TABLE categoria(
    categoria_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    categoria VARCHAR(25) NOT NULL,
    activa BOOL
) ENGINE=InnoDB;
```

```
CREATE TABLE producto(
    producto_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    categoria_id INT(11) NOT NULL,
    producto VARCHAR(255) NOT NULL,
    precio DECIMAL(7, 2) NOT NULL,
    descripcion BLOB,
    FOREIGN KEY (categoria_id)
        REFERENCES categoria(categoria_id)
) ENGINE=InnoDB;
```

*Cuando **el campo de una tabla hace referencia a la clave primaria de otra tabla**, se denomina "clave foránea" o "**foreign key**" (en inglés). Para poder utilizar claves foráneas, MySQL necesita utilizar sí o sí, el motor **InnoDB** ya que es el único con soporte para éstas.*

Como podrás observar, el campo de la tabla producto, que hace referencia a la clave primaria de la tabla categoría, se llama igual (categoria_id). Podría tener un nombre diferente, pero más adelante lo veremos. **Este campo, debe ser creado en la tabla, como cualquier campo común. La principal diferencia, radica en que debemos indicar que este campo, debe ser tratado como una clave foránea.**

Para ello, utilizamos la siguiente sintaxis:

```
FOREIGN KEY (nombre_de_la_clave_foranea)
    REFERENCES tabla_relacionada(nombre_de_la_clave_primaria)
```

En lenguaje humano, esto se leería como sigue:

```
FOREIGN KEY (FK)
la clave foránea es FK
    REFERENCES TABLA(PK)
    que hace referencia a la tabla TABLA(a través del campo PK)
```

***FK** es una **Foreign Key** (clave foránea) mientras que **PK** es una **Primary Key** (clave primaria)*

Esto significa que siempre que debamos relacionar un campo con otro, el campo relacionado deberá indicarse como Foreign Key mientras que el campo al cuál hace

referencia, deberá indicarse como Primary Key.

Luego, podremos obtener, desde la base de datos, el “objeto” producto, incluyendo los datos a los cuáles hace referencia. Para ello, utilizaremos la siguiente consulta:

```
SELECT   producto.*, categoria.*
FROM     producto INNER JOIN categoria USING(categoria_id)
```

Con `SELECT producto.*, categoria.*` estamos seleccionando todos los campos de la tabla producto y todos los campos de la tabla categoria. Mientras que con `FROM producto INNER JOIN categoria USING(categoria_id)`, estamos diciendo que:

```
FROM producto INNER JOIN categoria
```

Desde la tabla `producto` unida internamente a la tabla `categoria`

```
USING(categoria_id)
```

utilizando el campo `categoria_id`

Cómo comentamos anteriormente, una FK no necesariamente debe llevar el mismo nombre que la clave primaria a la cuál hace referencia. Podríamos, por ejemplo, haber creado nuestra tabla `producto` de la siguiente manera:

```
CREATE TABLE producto(
    producto_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    FK_categoria INT(11) NOT NULL,
    producto VARCHAR(255) NOT NULL,
    precio DECIMAL(7, 2) NOT NULL,
    descripcion BLOB,
    FOREIGN KEY (FK_categoria)
        REFERENCES categoria(categoria_id)
) ENGINE=InnoDB;
```

Pero en este caso, deberíamos modificar la sintaxis de nuestra consulta:

```
SELECT   producto.*, categoria.*
FROM     producto INNER JOIN categoria
         ON producto.FK_categoria = categoria.categoria_id
```

Es decir, que ya no podemos indicarle que utilice el campo homónimo en ambas tablas, sino, que para realizar esta unión interna se base en la condición de igualdad del valor de los mismo (campo foráneo y primario respectivamente).

12 Bases de datos en Python con MySQL

Introducción a bases de datos con Python

En Python, el acceso a bases de datos se encuentra definido a modo de estándar en las especificaciones de **DB-API**, que puedes leer en la **PEP 249**: http://www.python.org/dev/peps/pep-0249/. Esto, significa que independientemente de la base de datos que utilicemos, los métodos y procesos de conexión, lectura y escritura de datos, desde Python, siempre serán los mismos, más allá del conector.

En nuestro caso particular, utilizaremos MySQL, para lo cual, vamos a trabajar con el módulo **MySQLdb**.

A diferencia de los módulos de la librería estándar de Python, MySQLdb debe ser instalado manualmente. Para ello, ejecutaremos el siguiente comando:

```
sudo apt-get install python-mysqldb
```

Conectarse a la base de datos y ejecutar consultas

Para conectarnos a la base de datos y ejecutar cualquier consulta, el procedimiento consiste en:

1. Abrir la conexión y crear un puntero
2. Ejecutar la consulta
3. Traer los resultados (si de una selección se trata) o hacer efectiva la escritura (cuando se inserta, actualiza o eliminan datos)
4. Cerrar el puntero y la conexión

> ***Los resultados de una consulta** de selección, se reciben en una **tupla**, cuyos elementos, son otras tuplas, conteniendo el valor de cada campo seleccionado de la tabla, en el orden que han sido seleccionados.*

Una forma simple de acceder a bases de datos

```
import MySQLdb


DB_HOST = 'localhost'
DB_USER = 'root'
DB_PASS = 'mysqlroot'
DB_NAME = 'a'


def run_query(query=''):
    datos = [DB_HOST, DB_USER, DB_PASS, DB_NAME]

    conn = MySQLdb.connect(*datos)    # Conectar a la base de datos
    cursor = conn.cursor()            # Crear un cursor
    cursor.execute(query)             # Ejecutar una consulta

    if query.upper().startswith('SELECT'):
        data = cursor.fetchall()      # Traer los resultados de un select
    else:
        conn.commit()                 # Hacer efectiva la escritura de datos
        data = None

    cursor.close()                    # Cerrar el cursor
    conn.close()                      # Cerrar la conexión

    return data
```

Insertar datos

```
dato = raw_input("Dato: ")
query = "INSERT INTO b (b2) VALUES ('%s')" % dato
run_query(query)
```

Seleccionar todos los registros

```
query = "SELECT b1, b2 FROM b ORDER BY b2 DESC"
result = run_query(query)
print result
```

Seleccionar solo registros coincidentes

```
criterio = raw_input("Ingrese criterio de búsqueda: ")
query = "SELECT b1, b2 FROM b WHERE b2 = '%s'" % criterio
```

```
result = run_query(query)
print result
```

Eliminar registros

```
criterio = raw_input("Ingrese criterio p7 eliminar coincidencias: ")
query = "DELETE FROM b WHERE b2 = '%s'" % criterio
run_query(query)
```

Actualizar datos

```
b1 = raw_input("ID: ")
b2 = raw_input("Nuevo valor: ")
query = "UPDATE b SET b2='%s' WHERE b1 = %i" % (b2, int(b1))
run_query(query)
```

13 Corriendo Python Apps en la Web

En este capítulo, nos concentraremos en aprender, como servir aplicaciones Python en la Web, corriendo bajo Apache, pero sin utilizar un framework, ya que el objetivo de este curso es entender el proceso de razonamiento para obtener la respuesta a “cómo resolver necesidades puntuales”.

Introducción

Existen en el mercado, varios frameworks que nos permiten crear aplicaciones Python robustas, de manera rápida y servirlas a través de Internet, en un sitio Web tradicional.

Algunos de ellos, poseen una arquitectura particular, como es el caso de Django, que utiliza un patrón arquitectónico denominado MVT (model-view-template), basado en MVC (model-view-controller) pero que prescinde del motor de éste: es decir, del controlador. Otro framework muy robusto también, es Web2Py, quien se caracteriza por tener una curva de aprendizaje menor que la de Django.

Sin embargo, para crear aplicaciones Python con estos frameworks, algunos requisitos deben ser tenidos en cuenta:

- Para crear aplicaciones escalables y mantenibles, que guarden un diseño arquitectónico coherente, es imprescindible tener un excelente dominio de la programación orientada a objetos y amplios conocimientos sobre patrones arquitectónicos y patrones de diseño;
- Como todo marco de trabajo, poseen sus propios métodos así como una sintaxis y pseudo-lenguaje propios, los cuales demandan invertir un tiempo considerable en aprender a utilizarlos. Es decir, no se requiere "aprender a programar un nuevo lenguaje" ni mucho menos "aprender a programar con ellos", sino que por el contrario, lo necesario es "aprender a utilizarlos" (es como aprender a utilizar un nuevo software). Y esto, insume muchísimo tiempo para que el aprendizaje sea fructífero y el aprovechamiento del framework, beneficioso;
- Son frameworks muy robustos, pensados para el desarrollo de grandes aplicaciones. Por ello, debe considerarse la posibilidad de prescindir de ellos, cuando lo que se necesite, sea una aplicación liviana, ya que el consumo de recursos y el rendimiento, no estará compensado por la robustez del desarrollo.

Pero más allá de todo lo anterior, la mejor forma de entender un framework, es comprender el lenguaje en el que han sido desarrollados y la forma en la que éstos, han logrado llegar a resolver una necesidad: servir aplicaciones Python a través de la Web.

*En este capítulo, nos concentraremos en aprender, **como servir aplicaciones Python en la Web, corriendo bajo Apache, pero sin utilizar un framework,** ya que **el objetivo de este curso es entender el proceso de razonamiento para obtener la respuesta a "cómo resolver necesidades puntuales"**.*

Python bajo Apache

Como si de una receta de cocina se tratara, vamos a aprender **cómo servir aplicaciones Web con Python, utilizando el servidor Apache**.

No es mucha la bibliografía que puede encontrarse al respecto, pero sí, lo es bastante confusa y hasta incluso contradictoria. Por eso, en este curso, nos vamos a proponer mantener el espíritu de simplicidad de Python, encontrando la manera simple de hacerlo.

¿Qué necesitamos?

En principio, necesitamos hacer que Apache, incorpore un soporte para servir archivos Python. Para ello, necesitaremos habilitarle un módulo, que brinde este soporte.

Existen varios módulos de Apache que brindan soporte para correr archivos Python. Uno de los más populares es el módulo **`mod_python`**, que sin embargo, presenta algunos problemas (descriptos en http://docs.python.org/release/2.7/howto/webservers.html#mod-python) que pueden prevenirse, utilizando el módulo **`mod_wsgi`** (http://docs.python.org/release/2.7/howto/webservers.html#mod-wsgi) el cula utilizaremos en este curso.

1. Instalación de mod_wsgi en Apache

Para habilitar mod_wsgi en Apache, basta con instalar el paquete libapache2-mod-wsgi:

```
sudo apt-get install libapache2-mod-wsgi
```

2. Crear la estructura de directorios para nuestra aplicación

Primero, es importante saber, como va a funcionar nuestra aplicación y cómo va a interactuar vía Web.

Debemos tener un **directorio destinado a montar toda la aplicación**:

```
mkdir /home/yo/curso-python/trunk/python-web
```

Dentro de este directorio, vamos a **dividir su arquitectura en dos partes**:

1) Destinada al almacenaje de nuestra aplicación Python pura (será un directorio privado, no servido)

2) Destinada a servir la aplicación (directorio público servido) en el cuál solo almacenaremos archivos estáticos

```
mkdir /home/yo/curso-python/trunk/python-web/mypythonapp
mkdir /home/yo/curso-python/trunk/python-web/public_html
```

Dentro de nuestro directorio `mypythonapp`, almacenaremos entonces, todos los módulos y paquetes de nuestra aplicación Python, mientras que en `public_html`, estarán todos los archivos estáticos y será el único directorio al que se pueda acceder mediante el navegador Web.

Aprovecharemos este paso, para crear una carpeta, destinada a almacenar los logs de errores y accesos a nuestra Web App:

```
mkdir /home/yo/curso-python/trunk/python-web/logs
```

3. Crear un controlador para la aplicación

Todas las peticiones realizadas por el usuario (es decir, las URI a las cuáles el usuario acceda por el navegador), serán manejadas por un único archivo, que estará almacenado en nuestro directorio `mypythonapp`.

```
echo '# -*- coding: utf-8 -*-' > mypythonapp/controller.py
```

Este archivo `controller.py` actuará como un pseudo `FrontController`[2], siendo el encargado de manejar todas las peticiones del usuario, haciendo la llamada a los módulos correspondientes según la URI solicitada.

Dicho módulo, solo se encargará de definir una función, que actúe con cada petición del usuario. Esta función, **deberá ser una función WSGI `application` válida**. Esto significa que:

1. Deberá llamarse **`application`**
2. Deberá recibir dos parámetros: **`environ`**, del módulo os, que provee un diccionario de las peticiones HTTP estándar y otras variables de entorno, y la función **`start_response`**, de WSGI, encargada de entregar la respuesta HTTP al usuario.

```
def application(environ, start_response):
    # Genero la salida HTML a mostrar al usuario
    output = "<p>Bienvenido a mi <b>PythonApp</b>!!!</p>"
    # Inicio una respuesta al navegador
    start_response('200 OK', [('Content-Type', 'text/html; charset=utf-8')])
    # Retorno el contenido HTML
    return output
```

Más adelante, veremos como crear un ***Application WSGI Controller***, mucho más potente.

---

2 **Front Controller** – Patrón de diseño: http://en.wikipedia.org/wiki/Front_Controller_pattern

4. Configurar el VirtualHost

En la buena configuración de nuestro `VirtualHost`, estará la clave para correr nuestra aplicación Python a través de la Web.

Mientras que el `DocumentRoot` de nuestro sitio Web, será la carpeta pública, `public_html`, una variable del `VirtualHost`, será la encargada de redirigir todas las peticiones públicas del usuario, hacia nuestro `FrontController`. Y la variable que se encargue de esto, será el alias `WSGIScriptAlias`:

```
sudo nano /etc/apache2/sites-available/python-web
```

Una vez allí, escribimos el contenido del nuevo virtual host:

```
<VirtualHost *:80>
    ServerName python-web

    DocumentRoot /home/yo/curso-python/trunk/python-web/public_html
    WSGIScriptAlias / /home/yo/curso-python/trunk/python-web/mypythonapp/controller.py
    ErrorLog /home/yo/curso-python/trunk/python-web/logs/errors.log
    CustomLog /home/yo/curso-python/trunk/python-web/logs/access.log combined

    <Directory />
        Options FollowSymLinks
        AllowOverride All
    </Directory>
</VirtualHost>
```

Una vez configurado nuestro VirtualHost:

1) Habilitamos el sitio Web: **`sudo a2ensite python-web`**
2) Recargamos Apache: **`sudo service apache2 reload`**
3) Habilitamos el sitio en nuestro host: **`sudo nano /etc/hosts`** y allí agregamos la siguiente línea: **`127.0.0.1     python-web`**

A partir de ahora, si abrimos nuestro navegador Web e ingresamos la url http://python-web veremos la frase: Bienvenido a mi **PythonApp**.

Agregar un nuevo hostname a nuestro /etc/hosts nos permitirá seguir trabajando normalmente con nuestro localhost, sin que nuestras aplicaciones Python interfieran con otras, ya sean webs estáticas en HTML o dinámicas en PHP u otro lenguaje.

Utilizando `environ` para manejar peticiones del usuario

El diccionario `environ` del módulo `os`, nos provee de la URI solicitada por el usuario, a través de la clave `REQUEST_URI`. Valiéndonos de ella, podremos crear una Application WSGI Controller mucho más potente, que nos permita hacer switch de la petición, para saber a qué módulo llamar.

Para ello y a fin de poder manejar imports absolutos evitando inconvenientes, primero debemos crear un archivo **`__init__.py`** en `mypythonapp` y luego, agregar el path de nuestra aplicación en nuestro controller.py, para que Python busque allí nuestros módulos:

```
from sys import path
path.append('/home/yo/curso-python/trunk/python-web/')
```

Ahora, todos nuestros imports los podremos hacer con el *namespace* absoluto, desde `mypythonapp`. Por ejemplo:

```
from mypythonapp.mi_paquete import mi_modulo
```

Nuestra app, podría por ejemplo, tener un paquete llamado `sitioweb`. Dentro de este paquete, podría tener varios módulos, correspondientes a cada una de las secciones de nuestro sitio. Valiéndonos de la clave REQUEST_URI de environ, podríamos hacer un switch como el que sigue:

```
from sys import path
path.append('/home/eugenia/borrador/python-web/')

from mypythonapp.sitioweb import contacto, default


def application(environ, start_response):
    peticion = environ['REQUEST_URI']

    if peticion.startswith('/contacto'):
        output = contacto.formulario()
    elif peticion.startswith('/gracias'):
        output = contacto.gracias()
    else:
        output = default.default_page()

    start_response('200 OK', [('Content-Type', 'text/html; charset=utf-8')])
    return output
```

Importante: *siempre, tras hacer un cambio a tu aplicación,* ***debes reiniciar Apache para que los cambios se vean reflejados*** *de manera correcta:* **`sudo service apache2 restart`**

Al ingresar a http://python-app/contacto, el contenido mostrado será el retornado por la función `formulario()` del módulo `contacto` del paquete `sitioweb.`

Si en cambio, ingresáramos en http://python-app/gracias, veríamos el contenido retornado por la función `gracias()` del mismo módulo.

Y si la URI solicitada no fuese ni una ni otra, siempre se mostrará el contenido retornado por la función `default_page()` del módulo `default` del paquete `sitioweb.`

Ten en cuenta que ***ningún*** **`print`** *de tu app* ***será tenido en cuenta****. Todas las funciones de tu app, tendrán que* ***hacer un*** **`return`** ***del contenido*** *que desees mostrar al usuario, para que el Application WSGI Controller, se encargue de entregarlos a WSGI y éste, de mostrarlos al usuario.*

Descarga el ejemplo de este capítulo, ingresando en:

http://curso-python.eugeniabahit.com/sources/python-web.tar.gz

14 Enviando e-mails con formato HTML desde Python

Paquetes necesarios

Para poder enviar e-mail desde nuestro servidor (u ordenador local), en primer lugar, es necesario contar con un **MTA** (*Mail Transpor Agent* o “Agente de transporte de correo”).

Uno de los MTA más populares para sistemas UNIX-Like, es sin dudas, el famoso ***sendmail***.

Para dejar nuestro servidor u ordenador local, listo para enviar mensajes de correo electrónico a través de Internet, solo será necesario entonces, instalar `sendmail`:

```
sudo apt-get install sendmail
```

Envío de e-mail desde Python

Para enviar e-mails desde Python, éste nos provee smtplib, otro módulo de la librería estándar de Python, quien nos permitirá enviar mensajes de correo electrónico, incluso, en formato HTML.

Solo necesitaremos:

- Crear un **objeto `smtplib.SMTP`** el cuál recibirá como parámetro de su método constructor, el host (localhost)
- Crear un mensaje de correo
- Enviar el mensaje mediante una llamada al **método `sendmail`** del objeto SMTP.

Más fácil es mirando el código:

```
# -*- coding: utf-8 -*-
import smtplib

remitente = "Desde gnucita <ebahit@member.fsf.org>"
destinatario = "Mama de Gnucita <eugeniabahit@gmail.com>"
asunto = "E-mal HTML enviado desde Python"
mensaje = """Hola!<br/> <br/>
Este es un <b>e-mail</b> enviando desde <b>Python</b>
"""

email = """From: %s
To: %s
MIME-Version: 1.0
Content-type: text/html
Subject: %s

%s
```

```
""" % (remitente, destinatario, asunto, mensaje)

try:
    smtp = smtplib.SMTP('localhost')
    smtp.sendmail(remitente, destinatario, email)
    print "Correo enviado"
except:
    print """Error: el mensaje no pudo enviarse.
    Compruebe que sendmail se encuentra instalado en su sistema"""
```

Así de simple, enviamos un e-mail con Python:

```
import smtplib
```

Importamos el módulo smtplib.

Luego, definimos las variables necesarias para el envío del mensaje (remitente, destinatario, asunto y mensaje -en formato HTML-):

```
remitente = "Desde gnucita <ebahit@member.fsf.org>"
destinatario = "Mama de Gnucita <eugeniabahit@gmail.com>"
asunto = "E-mal HTML enviado desde Python"
mensaje = """Hola!<br/> <br/>
Este es un <b>e-mail</b> enviando desde <b>Python</b>
"""
```

A continuación, generamos el e-mail con todos los datos definidos anteriormente:

```
email = """From: %s
To: %s
MIME-Version: 1.0
Content-type: text/html
Subject: %s

%s
""" % (remitente, destinatario, asunto, mensaje)
```

Y finalmente, creamos un objeto smtp:

```
smtp = smtplib.SMTP('localhost')
```

Y realizamos el envío:

```
smtp.sendmail(remitente, destinatario, email)
```

Envío de e-mails a múltiples destinatarios

Para enviar un e-mail a múltiples destinatarios, solo será necesario **generar una lista con los destinatarios**:

```
destinatarios = ['Persona A <maildepersonaA>', 'Persona B <maildepersonaB>']
```

Agregar una dirección de respuesta diferente

Cuando generamos el e-mail, es necesario saber, que todo tipo de cabeceras válidas, pueden agregarse. Incluso Reply-To:

```
email = """From: %s
To: %s
Reply-To: noreply@algundominio.com
MIME-Version: 1.0
Content-type: text/html
Subject: %s

%s
""" % (remitente, destinatario, asunto, mensaje)
```